ABSTRACT

PERFORMANCE OF DIFFERENT STRATEGIES FOR ERROR DETECTION AND CORRECTION IN ARITHMETIC ENCODING

by

George F. Elmasry

Lossless source encoding is occasionally used in some data compression applications. One of these encoding schemes is the arithmetic encoding.

When data is to be transmitted via communication channel, noise and impurities imposed by the channel cause errors. To reduce the effect of errors, channel encoder is added prior to transmission through the channel. Channel encoder inserts some bits that help channel decoder at the receiver end to detect and correct errors. These added error detection and correction bits are redundancy that causes reduction in the compression ratio and hence an increase in data rate through the channel. The higher the detection and correction capability, the larger the added redundancy needed.

Different approach for error detection and correction is used in this work. It is suitable for lossless data compression wherein errors are assumed to occur with low rate but causes very high propagation. That is, an error in one data symbol causes all the following symbols to be in error with high probability. This was shown to be the case in arithmetic encoding and Lemple-Ziv algorithms for data compression.

With this approach, redundancy in a form of a marker, is added to the data before it is compressed by the source encoder. The decoder examine the data for existence of errors and correct them.

Different approaches for redundancy marker is examined and compared. As a measure for comparison, we used misdetection by testing one or more marker location, as well as miscortrection. These performance measures are calculated analytically and by computer simulation. The results are also compared to those obtained with channel encoding such as Hamming codes.

We found that our approach performs as well as channel encoder. However, while Hamming codes results in an erroneous data when more than one error occurs, this approach gives a clear indication for this situation.

PERFORMANCE OF DIFFERENT STRATEGIES FOR ERROR DETECTION AND CORRECTION IN ARITHMETIC ENCODING

by
George F. Elmasry

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Electrical Engineering

Department of Electrical and Computer Engineering

May 1993

APPROVAL PAGE

Performance of Different Strategies for Error Detection and Correction in Arithmetic Encoding

George F. Elmasry

Dr. Yeheskel Barness, Thesis Adviser — Date
Director of the Center of Communications and Signal Processing Research
Distinguished Professor of Electrical and Computer Engineering
NJIT

Dr. Zoran Siveski, Committee Member — Date
Assistant Professor of Electrical and Computer Engineering
NJIT

Dr. Yun-Qing Shi, Committee Member — Date
Assistant Professor of Electrical and Computer Engineering
NJIT

BIOGRAPHICAL SKETCH

Author: George F. Elmasry

Degree: Master of Science in Electrical Engineering

Undergraduate and Graduate Education:

- Master of Science in Electrical Engineering,
  New Jersey Institute of Technology, Newark, NJ, 1993

- Bachelor of Science in Electrical Engineering,
  Alexandria University, Alexandria, Egypt, 1985

Major: Electrical Engineering

TABLE OF CONTENTS

Chapter **Page**

LIST OF TABLES

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

Communication systems are becoming more and more sophisticated. When technology was simple, the defects and noise of the channel were simply overpowered by using a strong enough signal or a slow enough transmission rate. Nowadays, with the introduction of VLSI, very complex hardware became cheap, while power and bandwidth remained resources to be conserved. Channels remain impaired by noise, interference and other defects, and one wishes to transmit even more data through them. Hence there is a need for more powerful coding schemes.

1.1 What is Data Compression?

Contrary to the belief of many, the idea of data compression is not new. There has always been an interest in economical communication, whether it be oral or written, electromagnetic; analog or digital. There is still today a widespread use of abbreviations and acronyms in both oral and written materials. The Morse code, which made early telegraphy possible, is an example of an early data compression technique.

We can define data compression as the process of encoding a body of data D into a smaller body of data $\Delta\langle D\rangle$ such that it is possible for $\Delta\langle D\rangle$ to be decoded back to D or some acceptable approximation to D. The data compression techniques can be broadly classified into either reversible (or redundancy-reduction) or irreversible (or entropy-reduction) techniques.

The irreversible technique, which also is called data compaction, achieves compression by reducing the information and retaining only a subset of the message set. Quantization techniques, a form of data compaction, are used to encode a continuos source into a discreet source. Since some information is lost in the process, an exact replica of the

original message can never be reconstructed. Usually, data compaction is used only in applications where an approximation of the original message set is sufficient.

We are interested in the reversible technique, where it is possible to recover all the original data. If we think of data as a combination of information and redundancy, the reversible techniques encode the source data with a view to remove (or at least to reduce) the redundancy in such a way that it can be subsequently reinserted into the data. Hence, these are called reversible techniques.

1.2 Coding Schemes

The implementation of such schemes were first described by Shanon[1] and Fano[2]. An improved method was proposed by Huffman[3], who developed a procedure which yields minimum average word-length for encoding statistically independent sources. Huffman's solution for data compression, however, is unsatisfactory in some applications due to the complexity and inefficiency of the encoding and decoding operations. One major problem, which is of particular importance in the case of encoding, is the need to know the statistics of the source alphabet, or alternatively to scan the source data to gather the statistics.

Davisson[4] showed that an optimum source code can be designed without any knowledge of the statistical properties of the source. The theory of optimum codes for source with unknown statistics is called *universal coding* . An encoder is called *universal* if its performance, after being designed without prior knowledge of the source statistics, converges, as the block length approaches infinity, to the performance of an encoder with prior knowledge of the source statistics. The universal encoding algorithms are capable of estimating, either directly or indirectly, the source statistics with increasing accuracy as the source string is being encoded. Since the source statistics are being estimated *on the fly* , and requires only one pass over the data, these algorithms have important applications in compression for information transmission.

Some examples of the universal coding schemes are, the Lynch-Davisson[5] code in which the redundancy of the code converges to zero as the block length approaches infinity. Also the incremental parsing technique described by Lempel-Ziv [6][7][8] and its character extension suggested by Welch[9] and the arithmetic code of Langodon-Rissanen[10][11] are examples of universal coding schemes.

One of the main drawbacks of the universal coding algorithms (and this is true of most reversible data compression schemes) is their high error propagation in the event of occurrence of a channel error. That is an error in the compressed image is not only reflected in that part of the reconstructed data but also tends to affect much of what follows it. An error in a single bit can cause the loss of the self-synchronizing property of these codes, resulting in the loss of a large block of data. The amount of damage that an error causes to the reconstructed data, which can be used as a measure of error propagation, depends on the compression scheme used. The reader is referred to [19] for more information about error propagation. Since these schemes are so intolerant to errors, their use over noisy channels is often limited.

1.3 The Communication Channel

As the demand for communication capacity continues to grow in such communication areas as person-to-person, broadcast, intercomputer and intracomputer, engineers are pressed to improve performance by trying to maximize the transmitted information rate through an available communication channel. Figure (1-1) presents the communication problem where the source of information is to be connected to a user by a channel. A device is inserted between the source and the channel called the encoder, and another device is inserted between the channel and the user called the decoder.

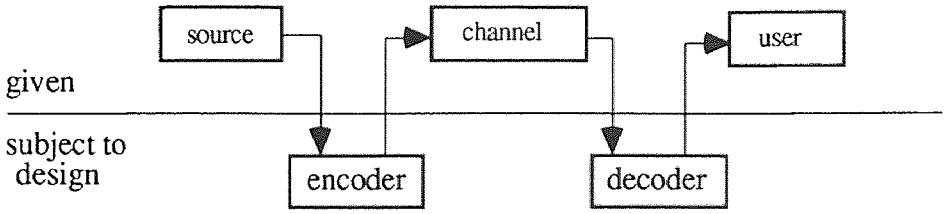

Fig. 1-1 The Communication Channel

In his study of the above communication problem, Shannon showed that a nearly error-free communication is possible over a noisy channel, provided an appropriate preprocessor called the encoder and an appropriate post processor called the decoder are allowed at each end of the communication link. However, he did not tell how to design the best encoders and decoders. Although considerable work has gone into an attempt to solve these problems, complete solutions are still unknown.

Fig(1-2) presents how the source is compressed (redundancy is removed) by the source encoder, then certain redundancy is introduced by the channel encoder for error correction. At the receiver the channel decoder correct errors and remove the redundancy, then the source decoder decompresses the data to the original information.

The main objective of this work is to study the idea of adding the redundancy needed for detection and correction before compression, and compare the results with the solutions offered by standard channel coding.

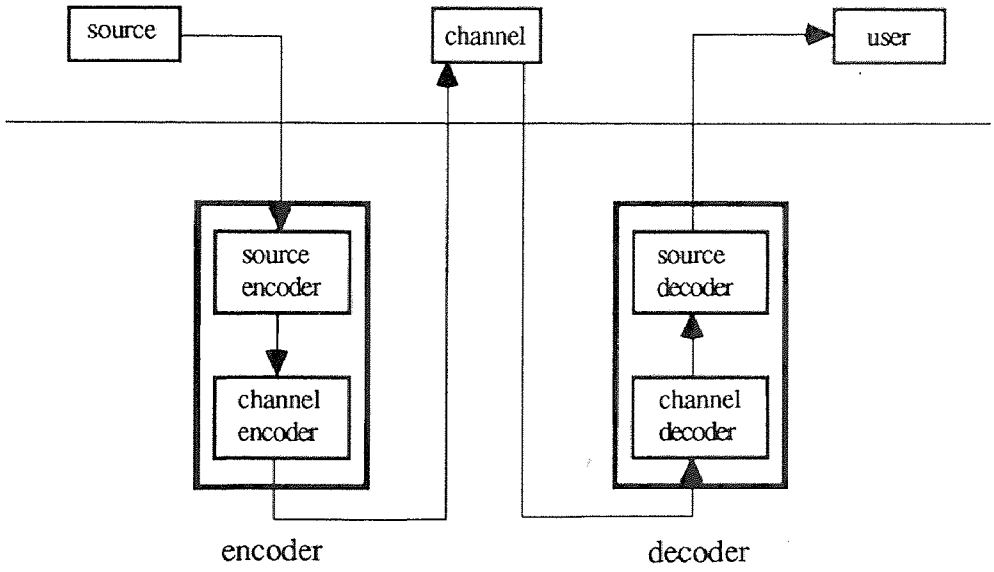

Fig. 1-2 The Communication Channel Problem

CHAPTER 2

DATA COMPRESSION

A discrete-time information source produces messages by emitting a sequence of symbols from a fixed alphabet called the source alphabet. The data can be highly redundant and hence waste the resources of the communication system. Data compression is a coding scheme used to represent the output of data source efficiently. The source output symbols is represented as a stream of bits suitable for transmitting through the channel. Data compression can considerably reduce the number of bits needed to represent the output sequence of a data source as compared with a simple binary representation of this source output.

2.1 Information Content and Entropy

A discrete memoryless source is a discrete random process{.., X_{-2}, X_{-1}, X_0, X_1, X_2, ..} where the X_L are independent, identically distributed random variables taking values of source alphabet $\{a_1, a_2, \ldots\ldots, a_c\}$ with probability distribution $\mathbf{p}=\{p(a_1), p(a_2), \ldots\ldots p(a_c)\}$. Any two sources with the same probability distribution on their outputs will present the same data compression problem because one can always rename symbols within the encoder and the decoder. Thus, for data compression, the source alphabet is unimportant; only the *size* of this alphabet and its *probability distribution* matter, so that one can use the symbol **p** as a handy name of the source it describes. Clearly, a source whose output is not random, is completely predictable from the past history and contains no information.

We can define the information content of a source as the number of bits per source symbol needed on the average by the best data compression code for this source. The information content of a memoryless source **p** is measured by the entropy H(**p**). The

entropy of a probabilistic source is equal to the average amount of information per symbol generated by that source. In other words, the entropy of a source is the average number of bits necessary to specify which symbol has been selected by the source.

If a source output a_j occurs with a probability $p(a_j)$, then the amount of information associated with the occurrence of output a_j is defined to be:

$$I(a_j) = - \log p(a_j) \tag{2.1}$$

when the logarithm is to the base 2, the information is measured in units of bits

If the probability of selecting the source symbol a_j is $p(a_j)$, then the information generated each time a symbol a_j is selected is, $- \log_2 p(a_j)$ bits. From the law of large numbers, the symbol a_j will be selected on the average $n\ p(a_j)$ times in a total of n selections, so the average amount of information obtained from n source output is:

$$- n\ p(a_1) \log_2 p(a_1) - \cdots\cdots - n\ p(a_j) \log_2 p(a_j) - \cdots\cdots - n\ p(a_c) \log_2 p(a_c) \tag{2.2}$$

To obtain the average amount of information per source output symbol, we divide by n, Therefore the average information, or the uncertainty, which is also termed the entropy H(p) is given by,

$$H(p) = - \sum_{j=1}^{c} p(a_j) \log_2 p(a_j) \qquad \text{bits/symbol} \tag{2.3}$$

The efficiency of any compression code is defined as

$$Efficiency = \frac{H(p)}{\overline{L}} \times 100 \tag{2.4}$$

where $\overline{L}$ is the average length of the code word and is given by

$$\overline{L} = \sum_{j=1}^{c} p(a_j) l(a_j) \tag{2.5}$$

where $\overline{L} \geq H$, and $l(a_j)$ is the length of the code corresponding to the symbol a_j.

2.2 Prefix Coding

Suppose that a source has alphabet eight, with probability distribution given by

$p_1 = p_2 = p_3 = p_4 = 1/32$

$p_5 = p_6 = 1/16$

$p_7 = 1/4$

$p_8 = 1/2$

The entropy H = 17/8 bits per symbol. The common method of representing a symbol of alphabet 8 source in binary requires three bits. A possible variable length code is illustrated in table (2.1)

Table 2-1 An Example of Prefix Coding

Symbol	Probability	code word	length
a	2^{-5}	00000	5
b	2^{-5}	00001	5
c	2^{-5}	00010	5
d	2^{-5}	00011	5
e	2^{-4}	0010	4
f	2^{-4}	0011	4
g	2^{-2}	01	2
h	2^{-1}	1	1

The average length of the code word is $\overline{L}$ = 17/8 bits per symbol which is equal to the entropy ($\overline{L} \geq H$ is satisfied). This code is a prefix code, i.e. there is no need to append extra symbols for punctuation. The code words can be run together without the possibility of ambiguity. Thus,

00011000000000001000101

can be decoded only as "dabch"

2.3 Huffman Coding

Huffman coding can be considered as a class of prefix coding. Consider a source with seven output symbol 'A, B, C, D, E, F and G' having the probabilities 3/8, 3/16, 3/16, 1/8, 1/16, 1/32 and 1/32 respectively. Figure (2.1) illustrates how the code construction proceeds. The original source is at the left side. At each step, as the tree is constructed to the right, the two symbols of smallest probability are combined, and the final tree is labeled with 0 and 1 arbitrarily at each branch. The code is illustrated in Table (2-2).

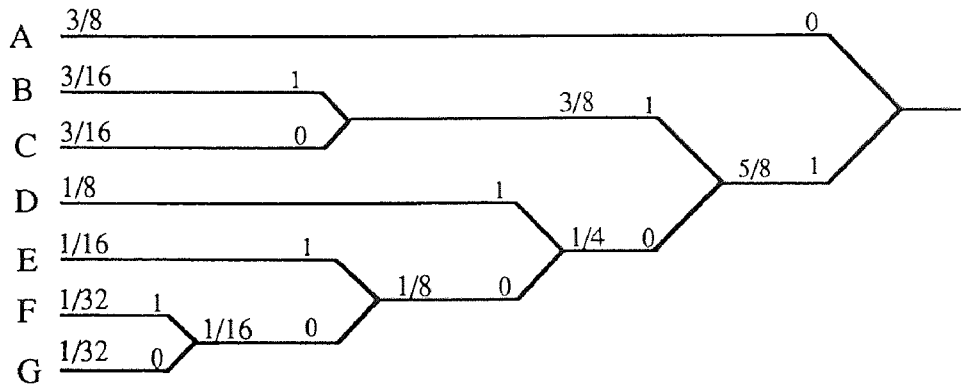

Fig. 2-1 The Code Construction of a Huffman Coding Example

Table 2-2 The Resultant code of a Huffman Coding Example

Symbol	Probability	Code word	Length
A	3/8	0	1
B	3/16	111	3
C	3/16	110	3
D	1/8	101	3
E	1/16	1001	4
F	1/32	10001	5
G	1/32	10000	5

for this example $\overline{L}$ = 2.44 while H = 2.37. This means that a small improvement can be obtained. Another example of Huffman coding, which leads to the next class of compression codes will be explained. Consider a source of alphabet three 'A, B and C' with probabilities 3/4, 3/16, and 1/16. From Figure (2-2), the Huffman code for single letters has code words 1, 01 and 00. The average block length is 1.25 while the entropy is 1.012, it is clear that a meaningful improvement is possible.

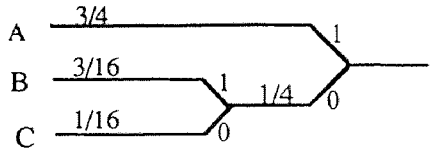

Fig. 2-2 A Single Letter Huffman Coding Example

Figure (2-3) shows the construction of the Huffman code for blocks of length 2 (extended Huffman Coding). The average length of the code word which encodes two source symbols is 2.09 bits. So, the rate of the code is 1.045 bits per source symbol. This should be compared with 1.25 bits per symbol, which is the rate of the simpler Huffman code, and with the 1.012 bits per symbol, which is the entropy of the source.

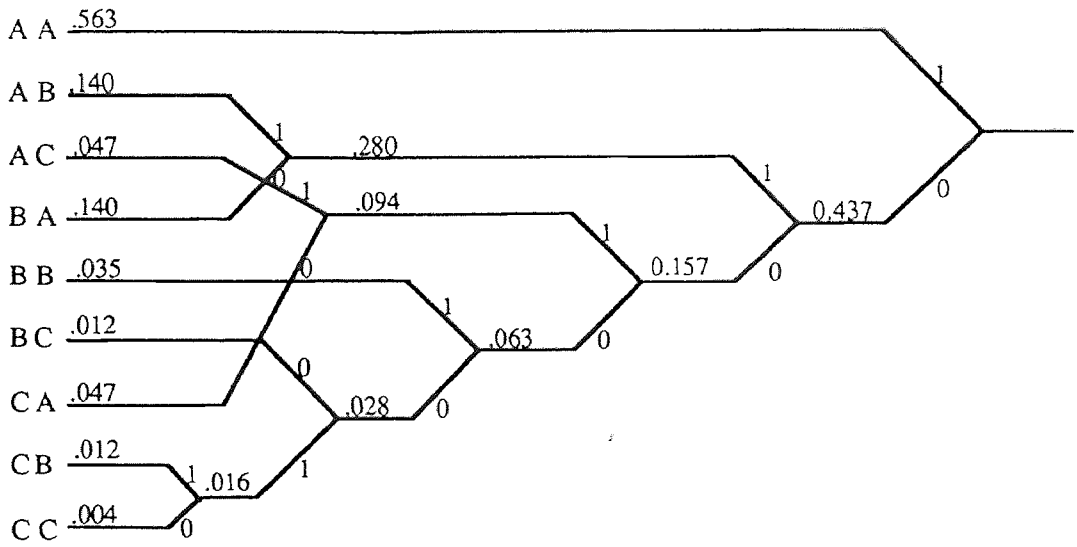

Fig. 2-3 An Example of Extended Huffman Coding

2.4 LZ Algorithm

The LZ algorithm converts variable-length strings of input symbols into fixed-length (or predictable length) codes. The symbol strings are selected such that, in the limit, all have almost equal probability of occurrence. Strings of frequently occurring symbols will contain more symbols than a string having infrequent symbols. This algorithm parses the source string into a collection of segments of gradually increasing length called the dictionary. There is no preference given to any particular symbol or segment. Starting with the empty segment, each new segment added to the collection is one symbol longer than the longest match so far found. For encoding, it is sufficient to transmit an index to the position of the longest match and the last added symbol.

Suppose that the source alphabet is binary with elements{0,1}. Initially the dictionary is empty. A source string 010111011 is parsed as {0, 1, 01, 11, 011}. When the parsed strings are retained in the same order in the dictionary, as they received, each segment can be encoded as the order pair (i, y), where the index i gives the position of the longest earlier found matching segment in the dictionary, and y gives the last added symbol. For example, the code for the segment 011 is the order pair (3, 1). It is easily seen that due to the nature of the incremental parsing algorithm, if any string is a member of the dictionary, then all its prefixes should also be members of the dictionary. This code is universal because the code length for an infinitely long source sequence converges to the entropy without any assumptions about the source probabilities.

The decoder reconstructs the segment corresponding to this pair and adds it to its dictionary. At any point of time the encoder and the decoder dictionaries are the same because they both use the same strategy to add segments to their individual dictionaries. The decoder is able to reconstruct the whole source string from the ordered pairs that are received. Table (2-3) shows the encoder and the decoder dictionaries, data sent and the decoded data for the example given above.

LZ algorithm is able to adapt the redundancy characteristics of the source, requiring no prior information about the source statistics. The LZ code is a variable-to-fixed length code (it encodes input string of variable length into codes of fixed length) unlike the Huffman code which is a fixed-to-variable length code (input strings of fixed length are encoded into variable length codes).

Some disadvantages of this algorithm are: A poor compression can result near the beginning of the file, hence it should not be used on short messages. The message should be long enough for the procedure to build enough symbol frequency experience to achieve good compression over the full message. Rapid changes in the redundancy characteristics of a very long file, may cause degradation in the compression achieved. The structure of the code word, an index and a row source symbol, may be inconvenient for a large source alphabet.

Table 2-3 An Encoding and Decoding Example of LZ Algorithm

encoder dictionary		$\{\phi,0\}$	$\{\phi,0,1\}$	$\{\phi,0,1,01\}$	$\{\phi,0,1,01,011\}$
sent	$(\phi,0)$	$(\phi,1)$	(0,1)	(1,1)	(01,1)
decoder dictionary	ϕ	$(\phi,0)$	$\{\phi,0,1\}$	$\{\phi,0,1,01\}$	$\{\phi,0,1,01,011\}$
decoded	0	1	01	11	011

2.5 LZW Algorithm

This algorithm retains the adaptive properties of the LZ algorithm and improves the compression ratio without sacrificing any of the simplicity of the data gathering process. Instead of treating the source as having a binary alphabet, an extended alphabet with 256 symbols is used. Hence, this algorithm is refereed to in the literature as the character extension improvement. Table (2-4) shows the dictionary, the sent string and the received string for the same example explained in the previous section.

In the original LZ algorithm, the code that is generated was an order pair consisting of an index and a source symbol. Thus the code contained some uncompressed data. Instead, the code generated by the LZW algorithm consists of a sequence of identifying numbers. This results in a significant improvement in compression of shorter source strings. Finally, while with the L-Z algorithm, the dictionary is initialized with the empty set, with LZW is done with all the alphabet characters.

Table 2-4 An Encoding and Decoding Example of LZW Algorithm

encoder						
dictionary	{0,1}	{0,1,01}	{0,1,01,10}	{0,1,01,10,011}	{0,1,01,10,011,11}	{0,1,01,10,011,11,101}
sent	0	1	01	1	10	11
decoder						
dictionary	{0,1}	{0,1,01}	{0,1,01,10}	{0,1,01,10,011}	{0,1,01,10,011,11}	{0,1,01,10,011,11,101}
decoded	0	1	01	1	10	11

Notice that LZW sends one code word each time, while LZ sends one code word plus row character each time.

CHAPTER 3

ARITHMETIC CODING

Arithmetic coding is a data compression technique that encodes data string by creating a code string which represents a fractional value on the number between 0 and 1. The coding algorithm is recursive, i.e. it operates on and encodes (decodes) one data symbol per iteration or recursion. On each recursion, the algorithm successively partitions the interval of the number line between 0 and 1, and retains one of the partitions which corresponds to the new string as a new interval. Thus, the algorithm successively generates smaller intervals, and the code string, viewed as a magnitude, lies in each of the nested intervals. The data string is recovered by using magnitude comparisons on the code string to recreate how the encoder must have successively partitioned and retained each nested subinterval.

This algorithm differs considerably from the more familiar coding techniques such as prefix (Huffman) codes. In Huffman coding, the file should be scanned to calculate the probability of each symbol before encoding, while arithmetic encoding is capable of accepting successive events from different probability distributions. Moreover the code acts directly on the probabilities, and can adapt *"on the fly"* to changing statistics.

3.1 Encoding

Let $A=\{a_1, a_2, \cdots\cdots, a_c\}$ be the source alphabet, with c different symbols, of a zero-memory information. Let this source emits symbol a_i with probability p_i. For each symbol a_i, we define the cumulative probability $P(a_i)$ by

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

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

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

Code Point The new left most 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.

$$New \quad C = current \quad C + current \quad W \,.\, P(a_i) \tag{3.2}$$

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:

$$New \quad W = Current \quad W \,.\, p(a_i) \tag{3.3}$$

where a_i is the current symbol.

When we start encoding, the initial values of code point is $C_0=0$, and the initial value of code interval is $W_0=1$

For the first source character s_1, to be encoded , we have

$$C_1 = C_0 + W_0 P(s_1) = P(s_1) \tag{3.4}$$

and

$$W_1 = W_0 p(s_1) = p(s_1) \tag{3.5}$$

For the second source character s_2,

$$C_2 = C_1 + W_1 P(s_2) \tag{3.6}$$

Substituting (3.4) , (3.5) in (3.6) we get ,

$$C_2 = P(s_1) + p(s_1) P(s_2) \tag{3.7}$$

After encoding the $K^{TH.}$ character in the source file, we have from equation(3.2),

$$C_k = C_{k-1} + W_{k-1} P(s_k) \tag{3.8}$$

and from equation(3.3),

$$W_k = W_{k-1} p(s_k) \tag{3.9}$$

But

$$C_{k-1} = C_{k-2} + W_{k-2} P(s_{k-1}) \tag{3.10}$$

and hence substituting equation (3.10) in equation (3.8), we get

$$C_k = C_{k-2} + W_{k-2} P(s_{k-1}) + W_{k-1} P(s_k) \tag{3.11}$$

Also

$$W_{k-1} = W_{k-2} p(s_{k-1}) \tag{3.12}$$

Substituting equation (3.12) in equation (3.9), we get

$$W_k = W_{k-2}\, p(s_{k-1})\, p(s_k) \tag{3.13}$$

Continuing in the same way equation (3.11) becomes

$$C_k = W_0\ p(s_1) + W_1\ p(s_2) + \cdots\cdots + W_{k-2}\ P(s_{k-1}) + W_{k-1}\ P(s_k) \tag{3.14}$$

and equation (3.13) becomes

$$W_k = W_0\, p(s_1)\, p(s_2) \cdots\cdots p(s_{k-1})\, p(s_k) \tag{3.15}$$

But $W_0=1$

$$W_k = p(s_1)\, p(s_2) \cdots\cdots p(s_{k-1})\, p(s_k) \tag{3.16}$$

Finally combining equations (3.14) and (3.16), we end up with,

$$C_k = P(s_1) + p(s_1)\, P(s_2) + p(s_1)\, p(s_2)\, P(s_3) + \cdots\cdots\cdots + p(s_1)p(s_2)\cdots\ p(s_{k-1})\, P(s_k) \tag{3.17}$$

Equation (3.17) can be rewritten as,

$$C_k = P(s_1) + p(s_1)[P(s_2) + p(s_2)[P(s_3) + \cdots\cdots [P(s_{k-2}) + p(s_{k-2})[P(s_{k-1}) + p(s_{k-1})P(s_k)]]..]] \tag{3.18}$$

Given that $P(s_k) < 1$, the term in the innermost parenthesis, implies

$$P(s_{k-1}) + p(s_{k-1})P(s_k) < P(s_{k-1}) + p(s_{k-1}) \tag{3.19}$$

But,

$$P(s_{k-1}) + p(s_{k-1}) = P(s_{k-1}+1) \tag{3.20}$$

where $s_{k-1}+1 = a_{j+1}$. if $s_{k-1} = a_j$ in the source alphabet.

Since $P(s_{k-1}+1) < 1$, then continuing in the same manner, we will have the outermost parenthesis is less than 1. Hence, from equation (3.18)

$$C_k < P(s_1) + p(s_1) \tag{3.21}$$

From equation (3.18), it is obvious that $C_k > P(s_1)$, therefore

$$P(s_1) < C_k < P(s_1) + p(s_1) = P(s_1+1) \tag{3.22}$$

Where $P(s_1)$ is the cumulative probability of the first source character to be encoded, and $P(s_1+1)$ is the cumulative probability of the symbol next to this character in the alphabet. That is if $s_1=a_i$, then $s_1+1= a_{i+1}$.

We 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 "$a_i\ a_i\ a_i$ ... ", then the code point is $P(a_i)$. On the other hand if the source string is "$a_i\ a_{i+1}\ a_{i+1}$ ... ", then the code point is very close to $P(a_{i+1})$, but not equal to $P(a_{i+1})$. Source strings begin with the source symbol a_i, are encoded to the interval $[a_i,a_{i+1})$ as shown from equation (3.22) $P(a_i) \leq C_k < P(a_{i+1})$.

3.2 Decoding

The first step in decoding 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; we subtract $P(s_1)$ from C_k :

$$C_k - P(s_1) = p(s_1)\,P(s_2) + p(s_1)\,p(s_2)\,P(s_3) + \cdots\cdots + p(s_1)p(s_2)\cdots\, p(s_{k-1})\,P(s_k) \qquad (3.23)$$

and divide $C_k - P(s_1)$ by $p(s_1)$

$$C_k^{(2)} = P(s_2) + p(s_2)\,P(s_3) + p(s_2)\,p(s_3)\,P(s_4) \ \ldots\ldots + p(s_2)p(s_3)\ldots\, p(s_{k-1})\,P(s_k) \qquad (3.24)$$

In the same manner we compare the new code point $C_k^{(2)}$ 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 is:

$$C_k^{(3)} = P(s_3) + p(s_3)\,P(s_4) + p(s_3)\,p(s_4)\,P(s_5) \ \ldots\ldots + p(s_3)p(s_4)\ldots\, p(s_{k-1})\,P(s_k)\,. \qquad (3.25)$$

In the same manner, reaching the last source character,

$$C_k^{(k)} = P(s_k) \qquad (3.26)$$

Which is the cumulative probability of the $K^{Th.}$ source symbol.

3.3 Example

Consider the alphabet 4 source, A={a, b, c, d}, with the relative frequencies 0.5, 0.25, 0.125 and 0.125 respectively. Table (3.1) shows the cumulative probabilities $P(a_i)$.

Table 3-1 The Cumulative Probabilities for an Arithmetic Coding Example

Symbol	Probability $p(a_i)$		Cumulative Probability $P(a_i)$	
	decimal	binary	decimal	binary
a	0.5	0.1	0	0
b	0.25	0.01	0.5	0.1
c	0.125	0.001	0.75	0.11
d	0.125	0.001	0.875	0.111

Figure (3.1) shows how the four code points divide the [0-1) interval into four subintervals. Notice that the code points are actually the sum of the probabilities of the preceding symbols for each symbol (cumulative probability).

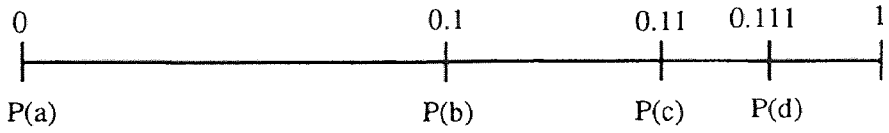

Fig. 3-1 The [0-1) Interval for an Arithmetic Coding Example

We identify each subinterval with its leftmost point and its width. For example the interval for the symbol "a" goes from 0 to 0.5, and for symbol "b" goes from 0.5 to 0.75. Notice that the width of each subinterval to the right of each code point corresponds to the probability of the symbol.

3.3.1 Encoding

Suppose we want to encode the following string " *acaabbda* " . The initial value of code point and code interval are ; $C_0=0$ and $W_0=1$

Applying equations (3.4) and (3.5) we get C_1 and W_1 in binary as follows

$$C_1 = 0 \quad \text{and} \quad W_1 = 0.1$$

Applying equations (3.2), (3.3) we have C_2 and W_2 in binary,

$$C_2 = 0.011 \quad \text{and} \quad W_2 = 0.0001$$

Finally when we reach the last symbol we will have,

$$C_8 = 0.0110001010111 \quad \text{and} \quad W_8 = 0.000000000000001$$

3.3.2 Decoding

We can consider the decoding procedure as the reverse of the encoding process. The decoder undo whatever the encoder does. For the same example given before, the decoder receives the encoded string 0.0110001010111. This code point is in the interval [0, 0.1), therefore the first decoded symbol is "a".

Since the accumulative probability of "a" is P(a)=0, and the probability of "a" is p(a)=0.1 , using equation (3.23) gives the code point for the next source character to be decoded.

$$C_k^{(2)} = \frac{0.0110001010111 - 0}{0.1} = 0.110001010111$$

This point is in the interval [0.11 , 0.111) , therefore the second decoded character is "c".

For the symbol "c" we have P(c)=0.11 and p(c)=0.001 , so the third code point is

$$C_k^{(3)} = \frac{0.110001010111 - 0.11}{0.001} = 0.001010111$$

This point is in the interval [0 , 0.1), therefore the third symbol to be decoded is "a" . but P(a)=0. and p(a)=0.1, so the fourth code point is

$$C_k^{(4)} = \frac{0.001010111 - 0}{0.1} = 0.01010111$$

The decoder proceeds in the same manner until the code point corresponds to the exact cumulative probability of the last encoded source character (see equation(3.26)). For further details with examples, refer to [10].

3.4 The Rounding Algorithm

In conventional arithmetic coding, as more symbols are included in the source sequences, the interval of decoding becomes more finely divided. The capacity of the decoder to accept more source symbols is limited by the ability of its fixed registers to resolve the boundaries between intervals. Any unanticipated rounding would be fatal to the processes of encoding and decoding. To avoid any inaccuracy which might be introduced by the use of floating point calculation, it is necessary to find an alternative representation of the probabilities p(s) by applying a scale factor u that converts the probability to frequency rate per u source symbols. To record these frequencies on a cumulative basis, the cumulative frequency table F is defined as

$$F_i = \left\lfloor \frac{1}{2} + u \sum_{1 \le j \le i} p_j \right\rfloor \quad , 0 \le i \le c \tag{3.27}$$

where $\lfloor x \rfloor$ is the large integer less than or equal to x.

Equation (3.27) gives $F_0 = 0$ and $F_c = u$. In practice the rounding effect provided by the equation can be avoided by suitable choice of the scale factor u. Table (3-2) shows an example of the same source alphabet used in section (3.3) with the symbol probabilities and the derived cumulative frequencies. Note that F_i is based on the actual frequency.

Table 3-2 An Example of the Cumulative Frequencies

Subscript i	Source Symbol a_i	Probability p_i	Cumulative Frequency F_i
1	a	0.5	500
2	b	0.25	750
3	c	0.125	875
4=c	d	0.125	1000=u

For more detailed information about how the boundaries of the intervals change due to rounding, the reader is referred to [23]. The final code word length is the same as for conventional arithmetic coding, eventhough the boundaries of the intervals deviate slightly as a result of rounding. This algorithm leaves the RADIX of the arithmetic (the alphabet of the encoded data) unspecified.

Although this algorithm is made for the applications of noiseless channel, in the next chapter, we will establish an algorithm for error detection and correction of the compressed data. The effect of RADIX will clearly be noticed in section(4.5.2). We will show the need to adapt this algorithm for the lowest possible RADIX .

CHAPTER 4

ADDING REDUNDANCY BEFORE COMPRESSION

As mentioned in chapter 1, compressed data can suffer very high error propagation in the event of channel error. An error in the compressed image does not affect only that part of the reconstructed data, but also affects much of what follows. An error in a single bit can cause the loss of self-synchronism and hence the loss of all data that follows. In this chapter, we will use the high error propagation property to establish error detection and correction schemes for different error rates.

By adding, as shown in Figure (4-1), a specific characters after each block of symbols of the source data before it is compressed, almost every error can be detected at the receiver by looking for that specific character marker at its position. To correct these errors, we resort on toggling each bit in the block until that specific marker shows up. We will use different types of marker strategies, and will study the expansion of the encoded data due to the added marker, as well as its effect on the system capability of detection and correction. Also, we will compare the results with the detection and correction obtained by conventional channel coding for the same error rate and show the advantages and disadvantages of the proposed scheme.

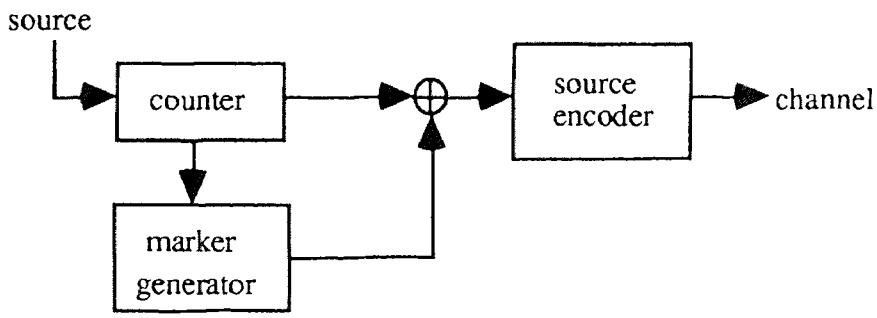

Fig. 4-1 Encoding Scheme

4.1 Decoding Procedure
With Error Detection and Correction

A register of length *bx* is inserted between the input encoded data and the source decoder as shown in figure(4-2). The required length of the register will be shown later to depend on both the error rate of the channel and the specific character marker used. We are assuming that at least the first *bx/2* characters in the file are error free. For a low error rate, this is rather non restrictive assumption, if the length of the register is chosen adequately.

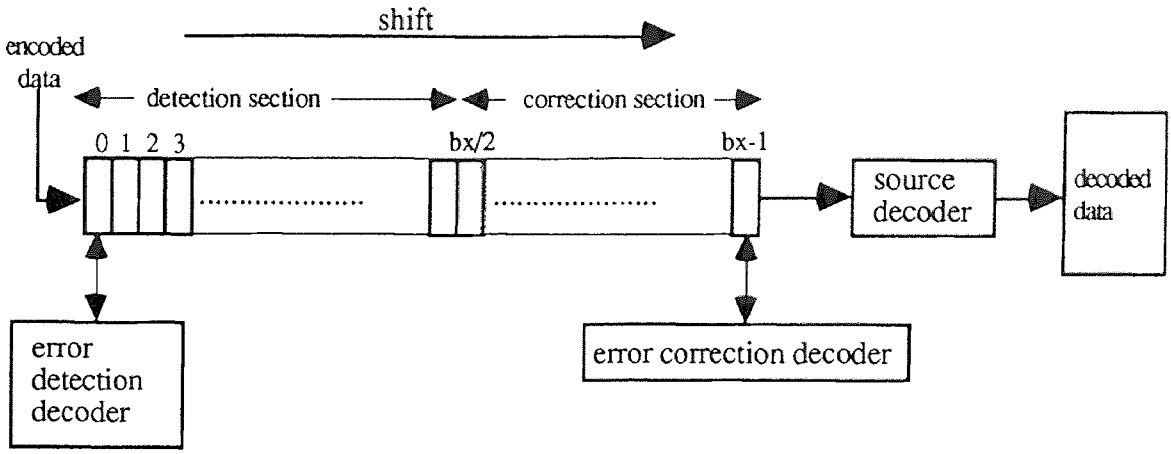

Fig. 4-2 Decoding Scheme with Detection and Correction

First the register is filled up with *bx* characters from the encoded data. While this is done, the error detection decoder decodes the character at position *0* and after each block of symbols, looks for the specific character marker at the decoded data. If the marker does not appear at the expected locations, this decoder sends a flag to indicate that there is an error, either at that point or before it; Due to the assumption that the first *bx/2* characters are error free, then, with probability one, the error is in the detection section of

the register in a location somewhere after the location where the previous marker did appear. With low probability we may detect a marker in its location eventhough an error has occurred before it. We term this as misdetection (or late detection) of the error. Therefore, if a marker did not appear at a certain location, error must have occurred between this location and the previous expected marker location, and with less probability between the previous marker location and the one before that, etc.

Clearly at any time, if the error detection decoder did not send the flag, either the character at position *0* is error free, or the decoder did not reach the marker position at the decoded data to check for error. Then a one character shift through the register takes place as follow:

1. The character at position *bx-1* is decoded by the source decoder.
2. The character at position *bx-2* is shifted to position *bx-1*, the character at position *bx-3* is shifted to position *bx-2*, and so on. Finally, the character at position 0 is shifted to position 1.
3. The next character in the encoded data is moved to position 0.

If the character at position *0* corresponds to an end of block so that a marker is expected at the decoded data, but the detection decoder does not show that, then an error is detected, which is assumed to be between position *0* and position *bx/2-1* as mentioned above. The error is shifted to the correction section of the register, between position *bx/2* and position *bx-1*, by applying the three steps mentioned above *bx/2* times.

To correct the error, every bit in the correction section is toggled starting from position *bx/2* . After toggling each bit, the whole register starting from position *bx-1* to position *0* is decoded by an error correction decoder. This decoder does the same function as the detection decoder, i.e. decoding and counting the decoded characters, and looks for the specific marker. If a wrong bit is toggled, the markers should not appear in their expected positions in the decoded data. That bit is switched back and the next bit is

toggled. On toggling the erroneous bit, all the markers in the reconstructed data should appear in their positions, and hence the error is considered to be corrected. It may happen that a wrong bit is toggled, that is error has not been corrected, but marker appears. this we term miscorrection which can happen with certain probability. In the next section, the probability of misdetection and the probability of miscorrection, will be discussed. we will also show how to overcome this problems. Figure (4-3) shows the flow chart of this scheme.

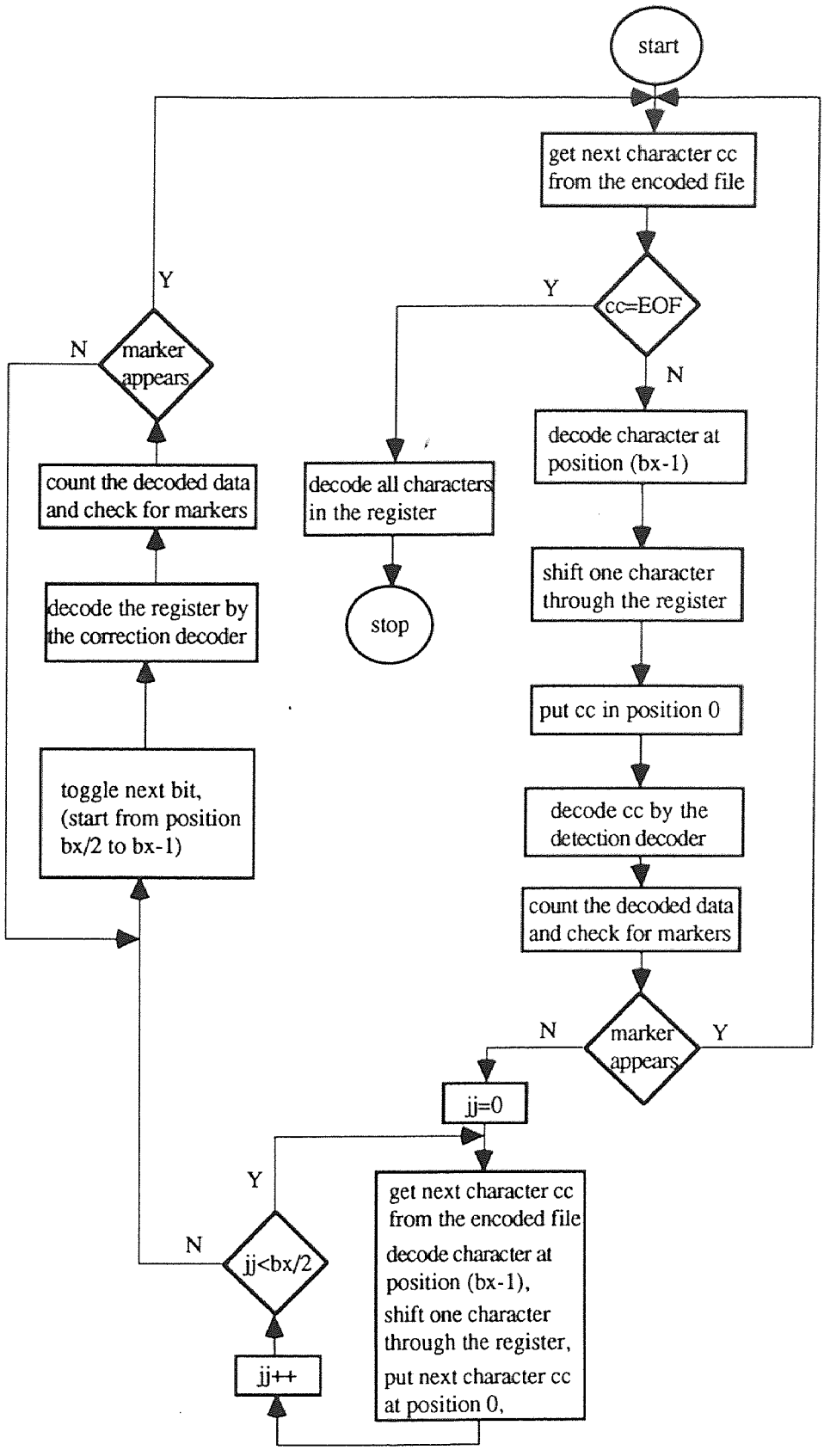

Fig. 4-3 The Flow Chart of the Proposed Scheme

4.2 Probabilities of Misdetection and Miscorrection

The probabilities of misdetection and miscorrection as well as the size of the file expansion depends on the marker strategy, the length of the marked block, and the specific file used. The following marker strategies will be examined:

1. *Particular character strategy*. Any character of the alphabet is used as a marker.
2. *Previous character strategy*. That is, the marker is chosen to be the last character of the block. For example, when we chose the block length to be eight characters the following data string:

   Today the weather is very good so we .....

   will become, when adding the marker:

   Today thhe weatheer is verry good sso we .....
3. *Average marker strategy*. Here we chose the marker a character which represents the ASCII integer less than or equal to the average of the ASCII values of all characters in the marked block, i.e.

$$S^{k+1} = \left\lfloor \frac{\sum_{i=1}^{k} S^i}{k} \right\rfloor \qquad (4.1)$$

where S^i is the ASCII value of the character i in the block, k is the block length, and S^{k+1} is the ASCII value of the marker. $\lfloor x \rfloor$ is the largest integer which is less than or equal to x

4.2 .1 Misdetection

Misdetection is defined as the probability not to detect an error in the expected location, given that an error took place. Let us assume that the error propagates totally in the decoded data, so that every character is in error with probability one. Therefore, each decoded character can be any character according to its probability in the alphabet.

With the *Particular character strategy* , let the marker character; a_n, has a probability of $P(a_n)$ in the alphabet. Then the probability of still having the marker a_n at the specific location, $P(a_n)$, is the probability of misdetection. For English text files this is given by the curve in Figure(4-4), where the x axis is the ASCII character value and the y axis is the probability of that character. Notice the peak at 32 which is the value of the space character.

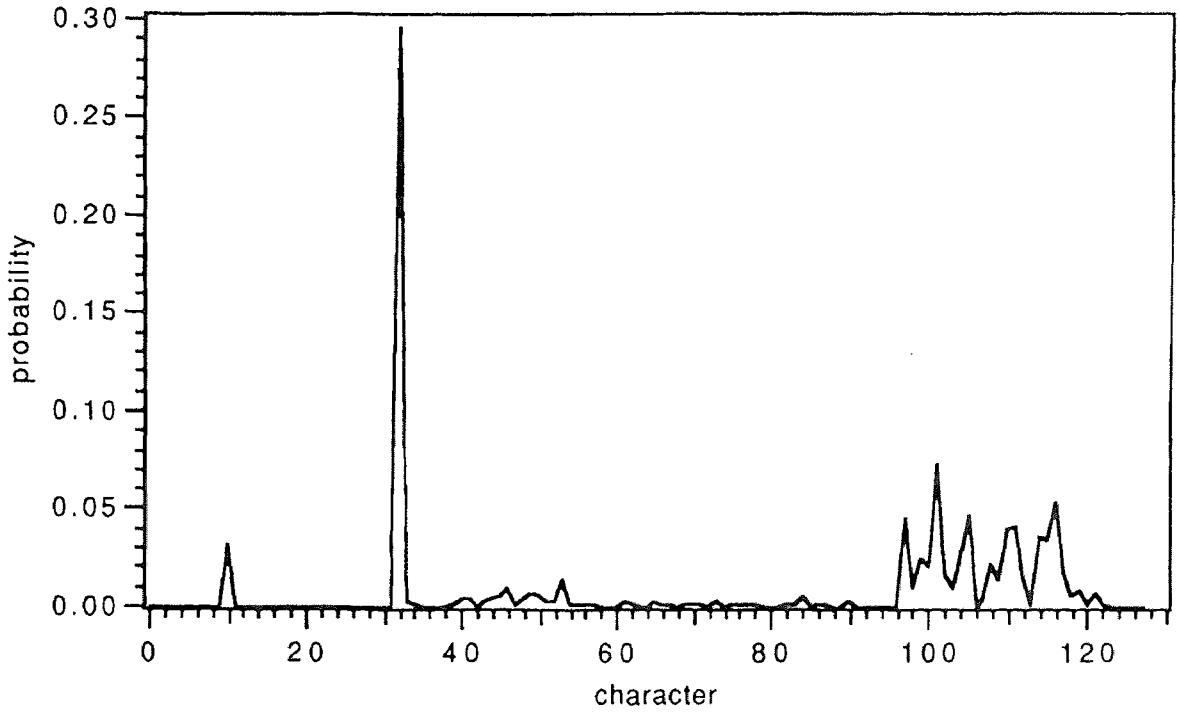

Fig. 4-4 Probability of Misdetection When Using a Particular Character as a Marker

Clearly, the smaller the probability $P(a_n)$ of the marker used, the lower the misdetection. Hence,

$$P_{\min} \leq P_1(mis\ \det ection) \leq P_{\max} \tag{4-2}$$

Where the probability of the most frequent character of the alphabet is on one extreme, and the probability of the least frequent character is on the other.

For the English text files,

$$0 \leq P_1(mis\ \det ection) \leq 0.295387$$

If all the alphabet characters are equiprobable, $P(a_n)=\frac{1}{M}$ for all a_n, then,

$$P_1(mis\ \det ection) = \frac{1}{M} \tag{4-3}$$

However, as we will show later, using a low probability character as a marker will increase the size of the compressed file more than using a high probability character. This means that there is a trade-off. Low probability character marker results in low probability of misdetection, but causes larger expansion in the compressed file and bigger loss in the file compression ratio. On the other hand, high probability character marker results in less expansion in the compressed file, but high probability of misdetection. One can remedy this conflict in the choice of marker by checking more than one location of marker and hence reduce the probability of misdetection to the order of $P^l(a_n)$ where l is the number of locations checked. This last approach of misdetection requires more complex hardware as well as software.

With the *Previous character strategy*, if we assume that the error propagates totally in the decoded data, then as mentioned before, each decoded character can be any

character in the alphabet according to its probability. Therefore, we have character a_n at the marker location with probability $P(a_n)$. But since we have to check whether the character at the marker location is the same as the previous character or not, the probability of misdetection is $P^2(a_n)$. Therefore,

P(misdetection / given character a_n is found at marker location and at the one before it)

$$= P^2(a_n)$$

and,

$$\overline{P_2(mis\,\det ection)} = \sum_{n=1}^{l} P(a_n) \,.\, [P^2(a_n)] \qquad (4\text{-}4)$$

where l is the size of the alphabet.

Figure (4-5) depicts the probability of the marker as a function of a_n, While figure (4-6) depict $P(mis\ \det ection)$ as a function of a_n. Clearly,

$$M P^3_{\min} \le \overline{P_2(mis\,\det ection)} \le M P^3_{\max} \qquad (4\text{-}5)$$

For English text files,

$$0 \le P_2(mis\ \det ection) \le 0.085562$$

and

$$\overline{P_2(mis\,\det ection)} = 0.026819$$

Where $P_{\max}$ and $P_{\min}$ are the probabilities of occurrence of the most and least probable marker in the alphabet. If all the alphabet characters are equiprobable, then,

$$\overline{P_2(mis\,\det ection)} = \frac{1}{M^2} \qquad (4\text{-}6)$$

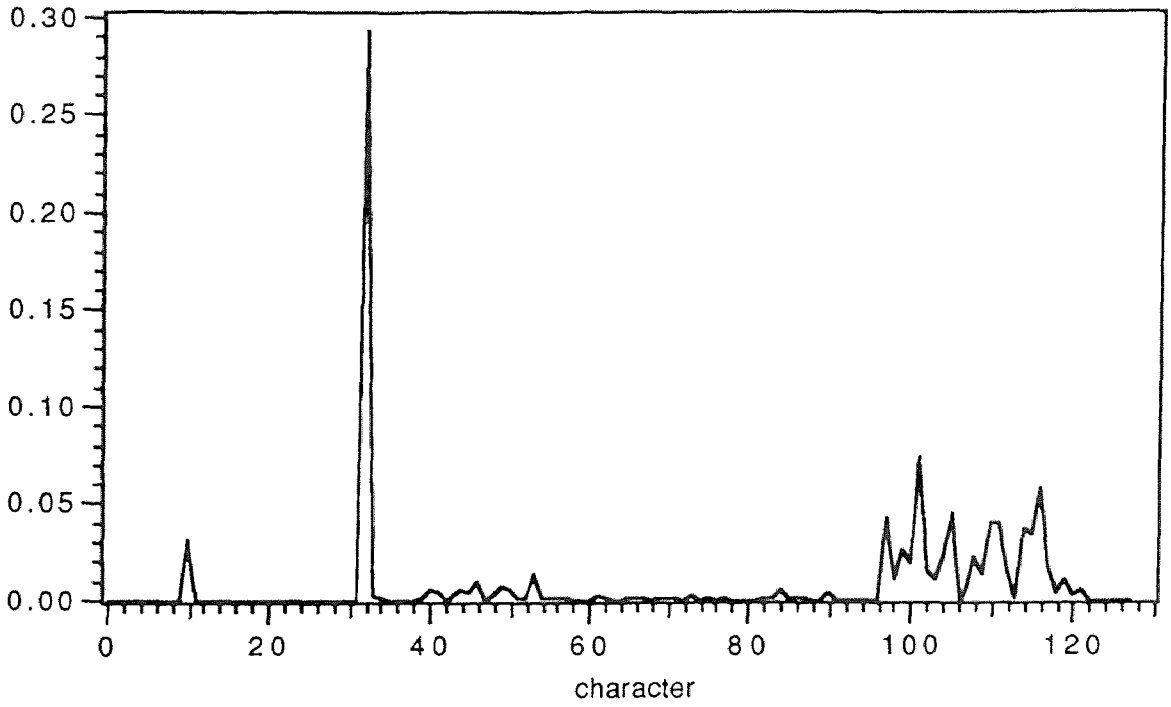

Fig. 4-5 Probability of the Previous Character as a Marker

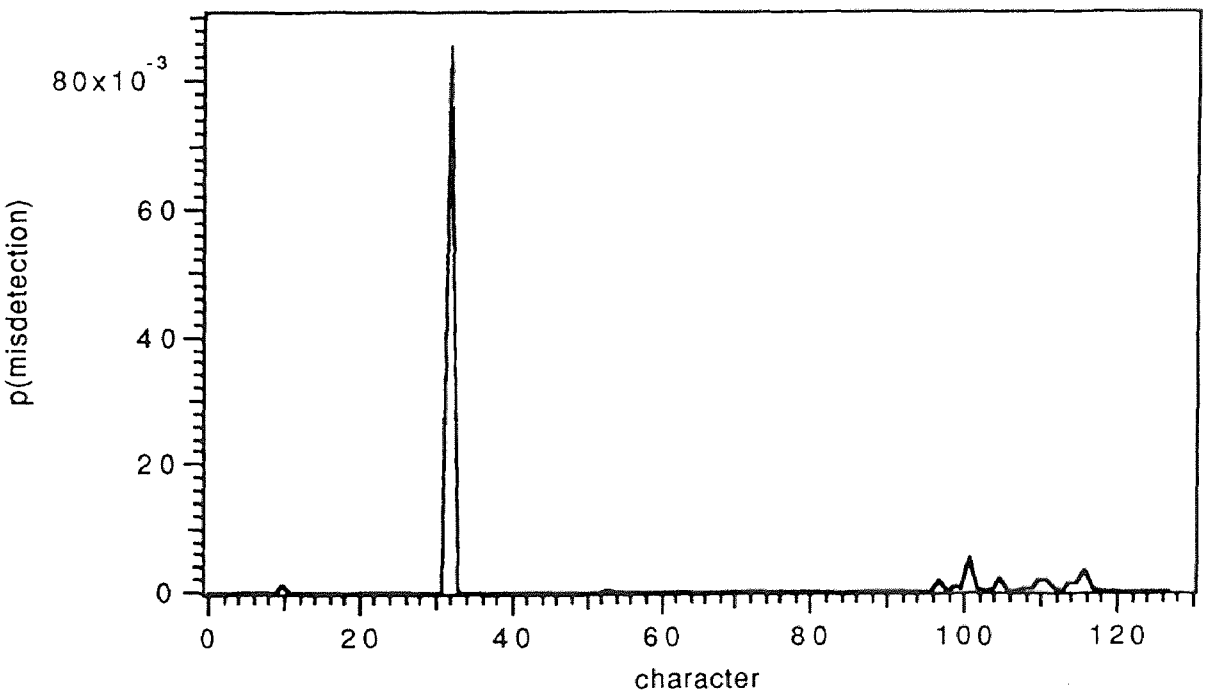

Fig. 4-6 Probability of Misdetection When Using the Previous Character as a Marker

With the *Average marker strategy* , under the same assumption of error propagation, misdetection happens if the character at the marker location represented in ASCII number equal the integer value of the average of the ASCII values of all characters in the previous block.

$$P_3(\text{misdetection} \;/\; S^{k+1} \text{ is the ASCI value of the marker}) = P\left(S^{k+1} = \left\lfloor \frac{\sum_{i=1}^{k} S^i}{k} \right\rfloor \right) \quad (4\text{-}7)$$

where S^{k+1} is the ASCII value of the character appears at marker location, S^i is the ASCII value of the character at location $i = 1,2, \cdots\cdots k$ of the previous block, k is the size of the block.

To calculate *P(misdetection)* analytically is quite complicated particularly for large k . Instead, based on ASCII files available to us, we calculated $P\left(S^{k+1} = \left\lfloor \frac{\sum_{i=1}^{k} S^i}{k} \right\rfloor \right)$ in the files. That is, we calculated the frequency of each character of the alphabet in the total markers generated in the files with this strategy. The files to be used for calculating these frequencies should be obtained from regular text files by fully randomizing the order of the characters in the files. If we assume that error propagate totally in the decoded data, then the aformated frequencies can equivalently obtained from the decoded data file after inserting an error. The probability curve of $P\left(S^{k+1} = \left\lfloor \frac{\sum_{i=1}^{k} S^i}{k} \right\rfloor \right)$ is depicted in figure (4-8). The curve in figure (4-7) shows the probability of each character S^i in these randomized files. From this we can calculate the probability of misdetection as follows,

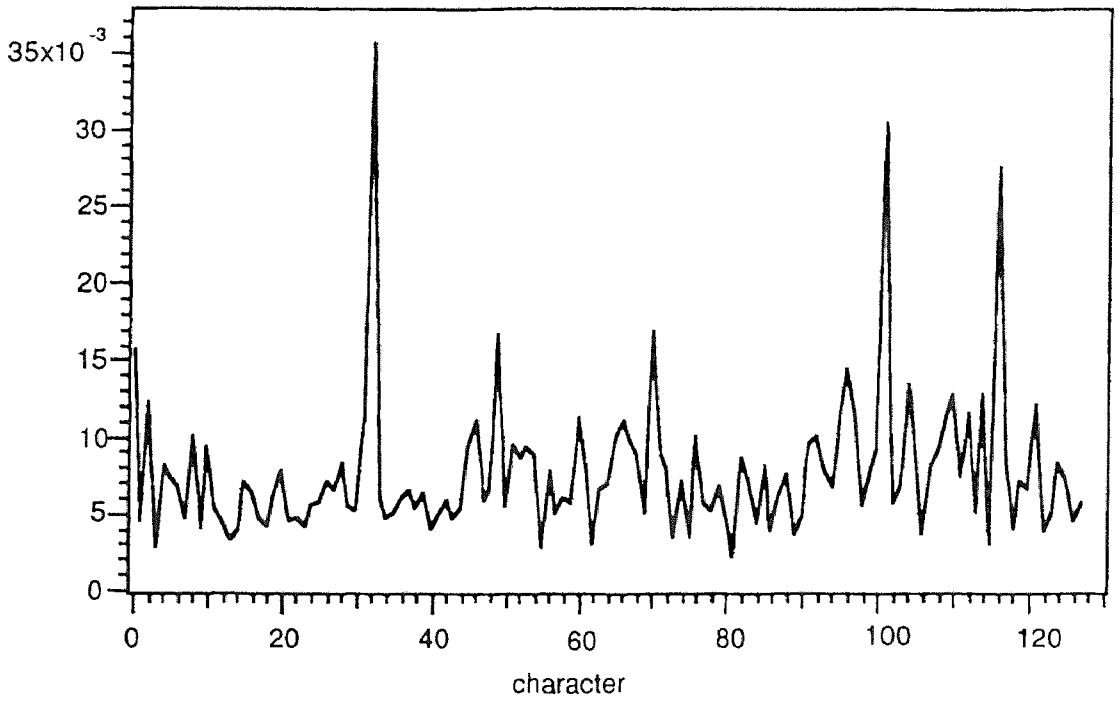

Fig. 4-7 Probability of Characters in Randomized Files

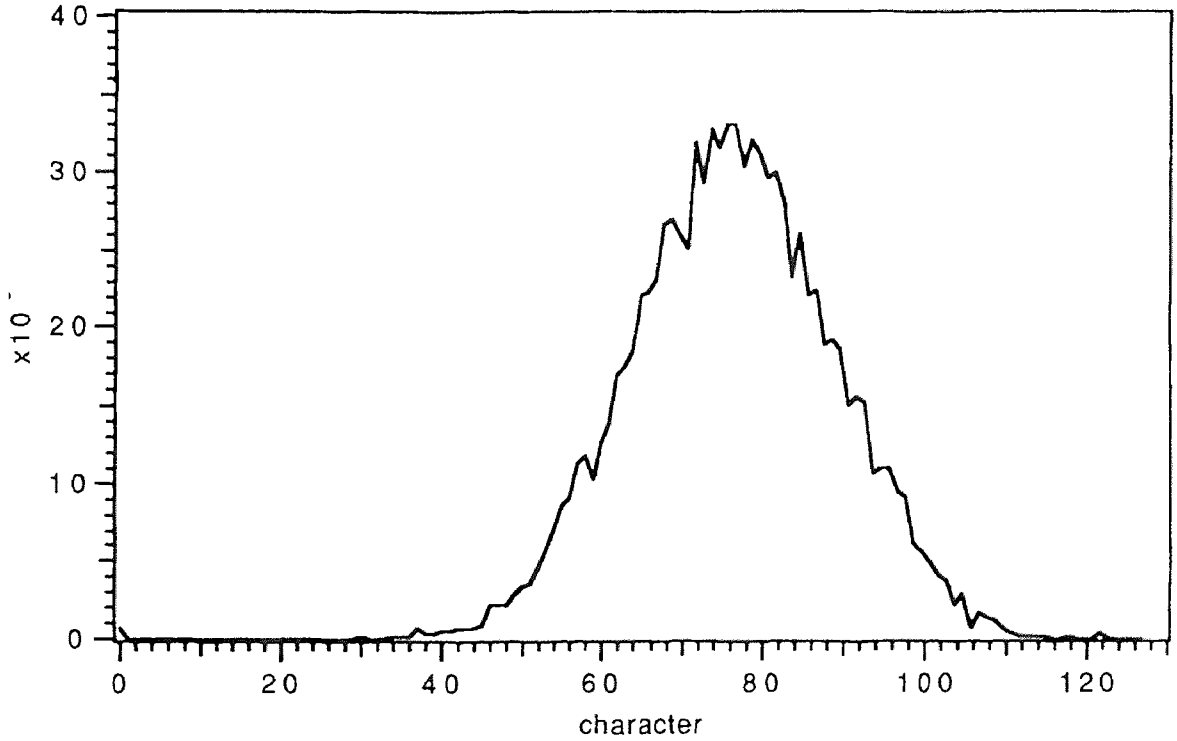

Fig. 4-8 Probability of Markers in Randomized Files

$$\overline{P_3(misdetection)} = \sum_{S_{k+1}=1}^{S_M} P(S^{k+1} = \left\lfloor \frac{\sum_{i=1}^{k} S^i}{k} \right\rfloor) P(S^{k+1}) \qquad (4\text{-}8)$$

Where S_M is the larger integer in the ASCII values of the alphabet.

Due to total randomization caused by the error, $P(S^{k+1})$ is the probability of S^{k+1} in the alphabet which is depicted in figure (4-7). From equation (4-8), the average probability of misdetection for the English text files is 0.007427.

4.2.2 Miscorrection

As stated in section (4.1), error correction is obtained by toggling the encoded data in the correction section of the register one bit at a time and looking for the marker in the reconstructed data. Every time a bit is toggled, there is a probability that the marker will appear eventhough a wrong bit is toggled. This can lead to a miscorrection.

With the *Particular character strategy* , with the same assumption of error propagation, let the marker character has a probability of $P(a_n)$ in the alphabet. Then the probability of misdetection when checking l markers is $P^l(a_n)$. If the number of marker locations checked for correction is $\bar{l}$, then the probability of miscorrection is a geometric distribution given as follow,

pr{miscorrection in toggling the 0th step}= $P^{\bar{l}}(a_n)$

pr{miscorrection in toggling the 1st step} = $\left[1-P^{\bar{l}}(a_n)\right]P^{\bar{l}}(a_n)$

pr{miscorrection in toggling the 2nd step} = $\left[1-P^{\bar{l}}(a_n)\right]^2 P^{\bar{l}}(a_n)$

pr{miscorrection in toggling the kth step} = $\left[1-P^{\bar{l}}(a_n)\right]^k P^{\bar{l}}(a_n)$

Hence, the probability of miscorrection is given by

$$P(miscorrection) = \sum_{k=0}^{\overline{N}} \left[1-P^{\bar{l}}(a_n)\right]^k P^{\bar{l}}(a_n) \qquad (4\text{-}9)$$

Where $\overline{N}$ is the average number of bits toggled before the erroneous bit is reached.

and the expected value of k is given by

$$E[k] = \frac{1-P^{\bar{l}}(a_n)}{P^{\bar{l}}(a_n)} \qquad (4\text{-}9a)$$

This means that, if we are using the space character as a marker which has a probability 0.295378, and we are checking 8 marker locations in correction, the expected value of k will be $1.73*10^4$. This number is the average number of toggling that leads to a miscorrection.

To reduce *P(miscorrection)*, we make $\bar{l} > l$. In Section(4-4) we will show how $\overline{N}$ is minimized to reduce *P(miscorrection)* . Minimizing $\overline{N}$ and making $\bar{l} > l$ stands also for the other two marker strategies.

4.3 Compressed File Expansion

Regardless of the marker strategy, adding a marker increases the size of the compressed file. Clearly the strategy which gives the smallest expansion is the better one.

4.3.1 With the Particular Character Strategy

Let us examine changing the entropy when adding a character as a marker. First notice that before addition we have,

$$H(p) = -\sum_{i=1}^{l} p(a_i) \log p(a_i) \tag{4-10}$$

Where l is the size of the alphabet, and the probability of the character a_i in the file is,

$$p(a_i) = \frac{N(a_i)}{L} \tag{4-11}$$

Where $N(a_i)$ is the total times of occurrence of a_i in the file, and L is the size of the file.

By adding a marker after each block of size k, the new probability of occurrence of each character in the alphabet will be,

$$p'(a_i) = \frac{N(a_i)}{L(1+1/k)} = \frac{p(a_i)}{1+1/k} \tag{4-12}$$

where a_i are characters which are not used as a marker. For the marker a_n,

$$p'(a_n) = \frac{N(a_n)+L/k}{L(1+1/k)} = \frac{p(a_n)+1/k}{1+1/k} \tag{4-13}$$

Hence the new entropy is given by,

$$H'(p) = -\sum_{i\neq n} \frac{p(a_i)}{1+1/k} \log \frac{p(a_i)}{1+1/k} - \frac{p(a_n)+1/k}{1+1/k} \log \frac{p(a_n)+1/k}{1+1/k} \tag{4-14}$$

$$\begin{aligned} H'(p) = & -\sum_{i\neq n} \frac{p(a_i)}{1+1/k} \log p(a_i) + \sum_{i\neq n} \frac{p(a_i)}{1+1/k} \log(1+\frac{1}{k}) \\ & - \frac{p(a_n)+1/k}{1+1/k} \log(p(a_n)+\frac{1}{k}) + \frac{p(a_n)+1/k}{1+1/k} \log(1+\frac{1}{k}) \end{aligned} \tag{4-15}$$

From(4-10),

$$H(p) = -\sum_{i\neq n}^{l} p(a_i) \log p(a_i) - p(a_n) \log p(a_n) \tag{4-16}$$

Also, the compression ratio of the file before adding the marker is,

$$R \leq \frac{\log_2 l}{H(p)} \tag{4-17}$$

After adding the marker, the compression ratio is,

$$R' \leq \frac{\log_2 l}{H'(p)} \tag{4-18}$$

In equation (4-17) and equation (4-18), the equality can be obtained for large files and effective compression algorithm.

Notice that $H'(p)$ is smaller than $H(p)$ which is expected due to an increase in the redundancy as a result of adding a marker. Therefore after adding a marker, the compression ratio increases. However, because of the expansion at the source file, the resulted compressed file is also expanded.

The size of the compressed file before adding the marker is given by,

$$\frac{L}{R} = \frac{L\,H(p)}{Log_2\, l} \quad \text{symbols} \tag{4-19}$$

After adding the marker,

$$\frac{L(1+1/k)}{R'} = \frac{L(1+1/k)\,H'(p)}{Log_2\, l} \quad \text{symbols} \tag{4-20}$$

Hence the extension of the file of size L is given by,

$$\frac{L(1+1/k)}{R'} - \frac{L}{R} = \frac{L}{Log_2\, l}\left[(1+1/k)\,H'(P) - H(p)\right] \quad \text{symbols} \tag{4-21}$$

Normalizing to L we get as a change in bit/symbols in compressed file to bit/symbol in uncompressed file,

$$\Delta = \frac{1+1/k}{R'} - \frac{1}{R} = \frac{1}{Log_2\, l}\left[(1+1/k)\,H'(P) - H(p)\right] \tag{4-22}$$

From equation(4-15),

$$\begin{aligned}(1+\frac{1}{k})\,H'(p) = &-\sum_{i\neq n} p(a_i)\log p(a_i) + \sum_{i\neq n} p(a_i)\log(1+\frac{1}{k}) \\ &- (p(a_n)+\frac{1}{k})\log(p(a_n)+\frac{1}{k}) + (p(a_n)+\frac{1}{k})\log(1+\frac{1}{k})\end{aligned} \tag{4-23}$$

From equations (4-16),(4-22) and (4-23) we get,

$$\begin{aligned}\Delta = \frac{1}{\log_2 l}\Big[&\sum_{i\neq n} p(a_i)\log(1+\frac{1}{k}) - (p(a_n)+\frac{1}{k})\log(p(a_n)+\frac{1}{k}) \\ &+ (p(a_n)+\frac{1}{k})\log(1+\frac{1}{k}) + p(a_n)\log p(a_n)\Big]\end{aligned}$$

$$\Delta \log_2 l = \log(1+\frac{1}{k}) \sum_{i \neq n} p(a_i) - (p(a_n)+\frac{1}{k}) \log(p(a_n)+\frac{1}{k})$$

$$+ \log(1+\frac{1}{k})\, p(a_n) + \frac{1}{k} \log(1+\frac{1}{k}) + p(a_n) \log(p(a_n))$$

$$\Delta \log_2 l = (1+\frac{1}{k}) \log(1+\frac{1}{k}) + p(a_n) \log p(a_n) - (p(a_n)+\frac{1}{k}) \log(p(a_n)+\frac{1}{k})$$

This equation gives the change in bit/symbol in the compressed file.

Finally dividing by $H(p)$, the compressed bit/symbol of the file before expansion, we get

$$\frac{\Delta \log_2 l}{H(p)} = \frac{1}{H(p)} \left[(1+\frac{1}{k}) \log(1+\frac{1}{k}) + p(a_n) \log p(a_n) \right.$$

$$\left. - (p(a_n)+\frac{1}{k}) \log(p(a_n)+\frac{1}{k}) \right] \qquad (4\text{-}24)$$

One can show easily from the curve in figure (4-9) that $p\log_2 p$ has a minimum of -0.53 at $p = 0.368$, That means it is a decreasing function of p for $p < 0.368$. Also if, $(p(a_n)+\frac{1}{k}) < 0.368$, then the last two terms in equation (4-24) is positive. Also, $\log(1+\frac{1}{k})$ is positive and hence we have always expansion despite the fact that $H'(p)$ decreases due to the increase of the redundancy in the expanded file. This is due to the increase in the size of the file after expansion.

If we use less probable character as a marker the sum of the two terms in equation(4-24) is larger as we are using the lower part of the $p\log_2 p$ curve. With larger k both the first and the last term in equation (4-24) becomes more positive.

Figure (4-10) depicts the dependencies of the percentage of the file expansion, as a function of the marker probability $p(a_n)$ when the block size k is taken as a parameter.

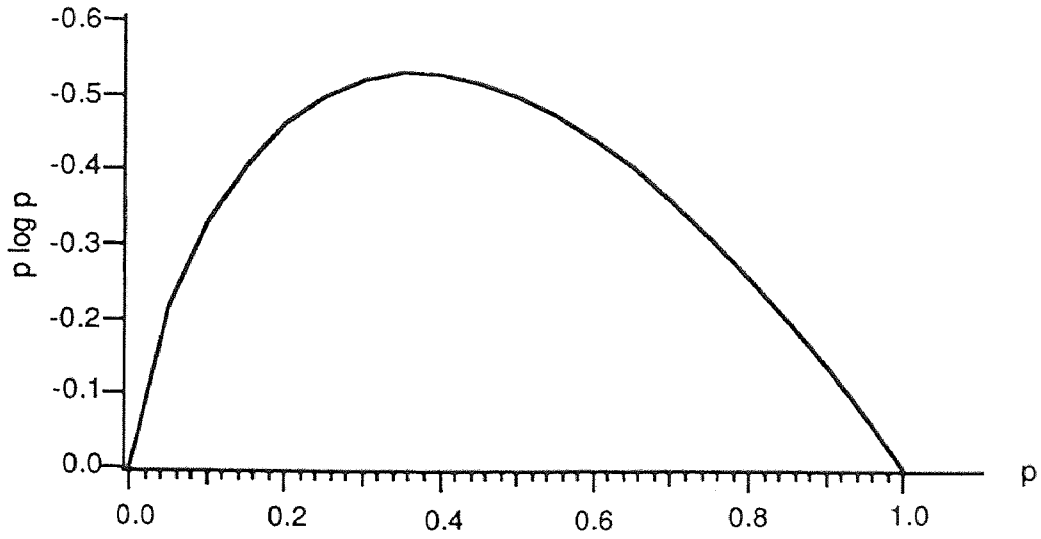

Fig. 4-9 *P log P* Versus *P* curve

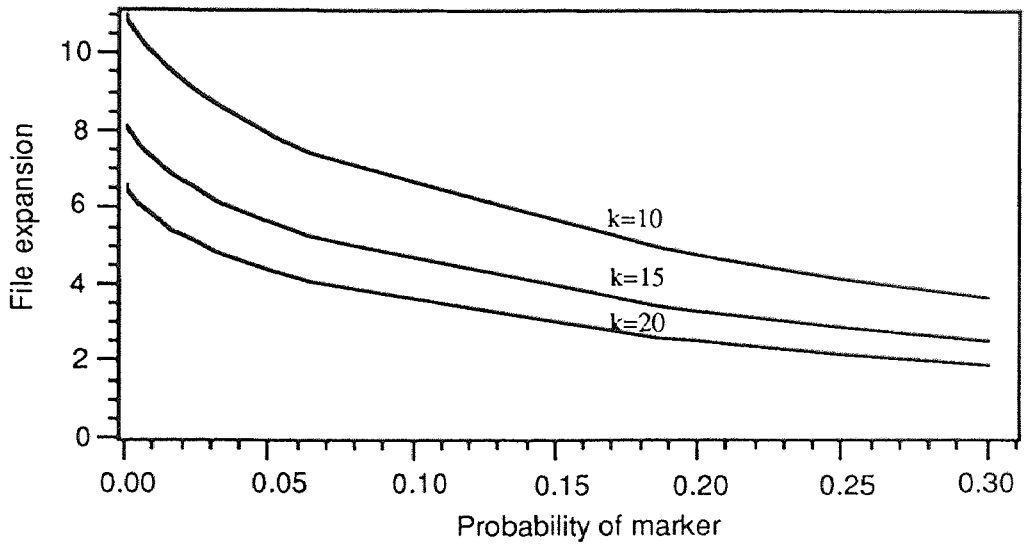

Fig. 4-10 The Percentage of File Expansion Related to the Probability of the Marker and the Length of the Block

4.3.2 With the Previous Character Strategy

The total number of each character changes according to its relative frequency in the file. Clearly, this is true provided that the file is large so that the probability of having the previous character to be a certain character of the alphabet is according to its relative frequency. Hence, using this definition, the new number of character a_i is,

$$N'(a_i) = N(a_i) + \frac{L}{k} p(a_i) = N(a_i)(1 + \frac{1}{k}) \qquad (4\text{-}25)$$

where i=1····· l

The new relative frequency is,

$$P'(a_i) = \frac{N(a_i)(1+1/k)}{L(1+1/k)} = P(a_i) \qquad (4\text{-}26)$$

That means the entropy does not change if the file is large enough so that the added character will have the same distribution as the original file and the entropy remains constant. The two curves in figure (4-4) and figure (4-5) shows that. Hence, equation (4-22) turns to,

$$\Delta = \frac{1+1/k}{R} - \frac{1}{R} = \frac{1}{Log_2 l} (\frac{1}{k}) H(p)$$

which makes the percentage of the file expansion,

$$\frac{\Delta \log_2 l}{H(p)} = \frac{1}{k}$$

i.e. the compressed file will increase by a factor $1/k$, where k is the block size.

4.3.3 With the Average Marker Strategy

In this strategy, we add a marker equal to the average character, that is,

$$S_{marker} = \left\lfloor \frac{\sum S^i}{k} \right\rfloor \tag{4-27}$$

where S^i is the ASCII value of the i^{th} character in the block.

The probability of S_{marker} is different from $p_3(misdetectction)$ of equation (4-7) in the fact that here the block has a certain dependency structure as in English text file, while there, they are totally random due to the error. Again due to the rounding and unknown text dependency structure, analytical calculation of S_{marker} is quite complicated. Instead we use many English text files to get $p(S_{marker})$ for marker as in figure(4-11).

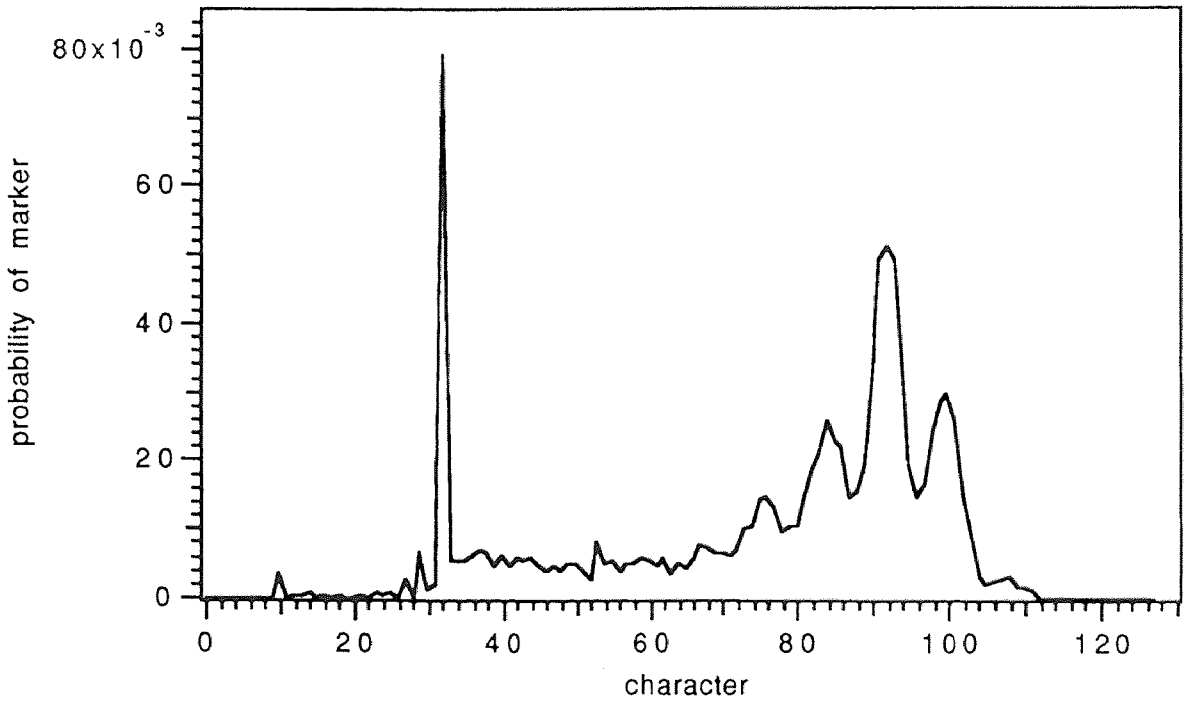

Fig. 4-11 Probability of Marker When Using the Average Marker Strategy

Figure (4-11) shows $p(S_{mar\,\ker})$ for marker being different character of the alphabet. Clearly, after expansion,

$$N'(a_i) = N(a_i) + \frac{L}{k} p\left(S_{mar\,\ker}=S(a_i)\right) \tag{4-28}$$

where $S(a_i)$ is the ASCII value for a_i

$$P'(a_i) = \frac{N(a_i)+\frac{L}{k}p\left(S_{mar\,\ker}=S(a_i)\right)}{L(1+1/k)} = \frac{p(a_i)+\frac{1}{k}p\left(S_{mar\,\ker}=S(a_i)\right)}{1+1/k} \tag{4-29}$$

Notice that when the marker is a particular character, $p\left(S_{mar\,\ker}=S(a_n)\right)=1$ for a certain n, and zero for all the others. Hence equation (4-29) reduces to that with particular character strategy, see equations (4-12), (4-13).

When the marker equal the previous character, then in equation (4-29) $p\left(S_{mar\,\ker}=S(a_i)\right)=p(a_{mar\,\ker} = a_i)$ for all i and hence equation (4-29) reduces to that obtained with previous character strategy, see equation (4-26).

The entropy after adding marker

$$H'(p) = -\sum_{i=1}^{l} \frac{p(a_i)+\frac{1}{k}p\left(S_{mar\,\ker}=S(a_i)\right)}{1+1/k} \log \frac{p(a_i)+\frac{1}{k}p\left(S_{mar\,\ker}=S(a_i)\right)}{1+1/k}$$

$$(1+\frac{1}{k})H'(p) = -\sum_{i=1}^{l}\left(p(a_i)+\frac{1}{k}p_{mi}\right) \log \frac{p(a_i)+\frac{1}{k}p_{mi}}{1+1/k}$$

Where $p_{mi} = p\left(S_{mar\,\ker}=S(a_i)\right)$ is given in figure(4-11)

$$(1+\frac{1}{k})\,H'(p) = -\sum_{i=1}^{l}\left(p(a_i)+\frac{1}{k}p_{mi}\right)\log\left(p(a_i)+\frac{1}{k}p_{mi}\right) + \log(1+\frac{1}{k})\sum_{i=1}^{M}p(a_i)$$

$$+\frac{1}{k}\log(1+\frac{1}{k})\sum_{i=1}^{M}p_{mi}$$

but $\sum_{i=1}^{l}p(a_i)=1$ and $\sum_{i=1}^{l}p_{mi}=1$, therefore,

$$(1+\frac{1}{k})\,H'(p) = -\sum_{i=1}^{l}\left(p(a_i)+\frac{1}{k}p_{mi}\right)\log\left(p(a_i)+\frac{1}{k}p_{mi}\right) + (1+\frac{1}{k})\log(1+\frac{1}{k}) \tag{4-30}$$

From equation (4-22),

$$\Delta = \frac{1}{Log_2\, l}\left[(1+1/k)\,H'(P) - H(p)\right]$$

$$\Delta = \frac{1}{\log_2 l}\left[\sum_{i=1}^{l}\left[p(a_i)\log p(a_i) - \left(p(a_i)+\frac{1}{k}p_{mi}\right)\log\left(p(a_i)+\frac{1}{k}p_{mi}\right)\right] + (1+\frac{1}{k})\log(1+\frac{1}{k})\right] \tag{4-31}$$

Equation (4-31) gives the change in bit/symbol in the compressed file to bit/symbol in the uncompressed file (compare this with equation (4-24) for the previous character). Clearly in equation(4-31) Δ is positive because each term is positive. But since $\frac{1}{k}p_{mi}$ is very small, the term in the summation will be small.

The curve in figure (4-10) depicts p_{mi} as a function of a_i and the curve in figure (4-4) depicts $p(a_i)$. Hence, $\frac{\Delta \log_2 l}{H(p)}$ which is the percentage of the file expansion is calculated and it equals 18.876577 for k=10.

4.4 Simulation and Results

4.4.1 With the Space Character Strategy

For any text file, the space character plays a major rule since it is the most frequent character used. The curve in figure (4-4) shows that the probability of the space character in the average for the text files used is 0.2957. Hence, using the space character as a marker results in small file expansion. Also making the marked block longer reduces the redundancy added as depicted in figure (4-9). Table (4-1) shows simulation results of the expansion ratio for different text files when using space character as a marker after a 20 characters long block.

Table 4-1 The Percentage of File Expansion for Different Text Files When Using the Space Character as a Marker After a Block of 20 Characters

file number	compressed file length at no marker	compressed file length with marker	redundancy added %
file (1)	8781	8940	1.8
file (2)	14251	14526	1.9
file (3)	8301	8438	1.6
file (4)	16363	16679	1.9

The simulation of this strategy showed that we need 60 characters in the register for detection. However, the probability of miscorrection calculated in equation (4-9) is relatively high, because increasing the block length increases $\overline{N}$, also$P(a_n)$ is high. This makes the number of marker locations needed for correction l' large. To approach a very low probability of miscorrection, we made the register length 150 characters, so that

when an error was detected, we shifted it to the correction section by applying the three steps mentioned in section (4-1) 90 times. This means shifting the error to a position between 90 and 150 in the register. Hence, we are checking all the markers in the decoded data corresponding to the characters between positions 0 and 90 (at least) for error correction.

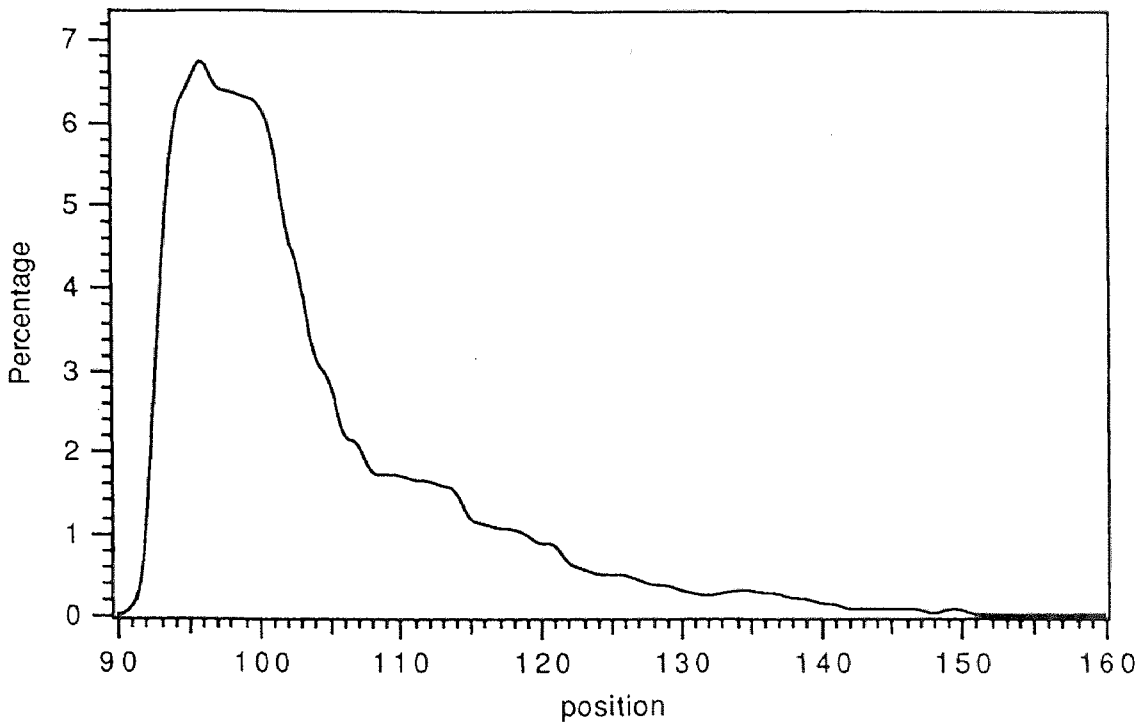

Fig. 4-12 The Relative Frequency of Correction at Each Position of the Correction Section of the Register With the Space Character

The experimental curve in figure (4-12) shows the relative frequency of correction at each position in the correction section of the register. The x axis shows the position at the register where the error is corrected, while the y axis shows the percentage of correction at this position. To reduce the average number of bits toggled before the erroneous bit is reached, $\overline{N}$ in equation (4-9), we start toggling from position 90 going to position 150 because errors are concentrated between positions 91 and 105.

This scheme is capable of correcting one error every 1200 bits since the register length is 1200 bits long. Because most of errors are detected at the first marker, errors at closer pattern can be corrected. For example, if the error being corrected is at position 100, the next error can be corrected if it is only 800 bits far apart.

To compare these results with channel coding, we consider the Hamming code (1023, 1013,1) which is capable of detecting and correcting one error every 1023 bits and the redundancy added is 1%. In the proposed scheme, as Table (4-1) shows, a redundancy of about 1.8% in the average is added. Here, we are capable of correcting one error and detecting as many errors as exist in the register. In Hamming code, if there is more than one error in the encoded block (the code word), the decoder decodes a different word which causes the loose of self synchronism in the reconstructed data.

4.4.2 With the Previous Character Strategy

In the simulation of this strategy, we make the register length 40 characters, and the block length 10 characters. For different English text files the expansion ratio of the compressed file is 10% as it was calculated in section (4.3.2). The probabilities of miscorrection and misdetection are insignificant. When an error is detected, the three steps mentioned in section (4.1) are applied 20 times so that the erroneous bit is moved to a position between 20 and 39 in the correction section of the register. Then we start toggling from position 20 going to position 39. The experimental curve in figure (4-13) shows the percentage of correction at each position in the correction section of the register. Note that the frequency between positions 22 and 29 is relatively high because most of the errors are detected at the first marker. Between positions 30 and 37 the frequency is small with gradual decay because these errors are detected at the second marker . In positions 38 and 39 the frequency is insignificant.

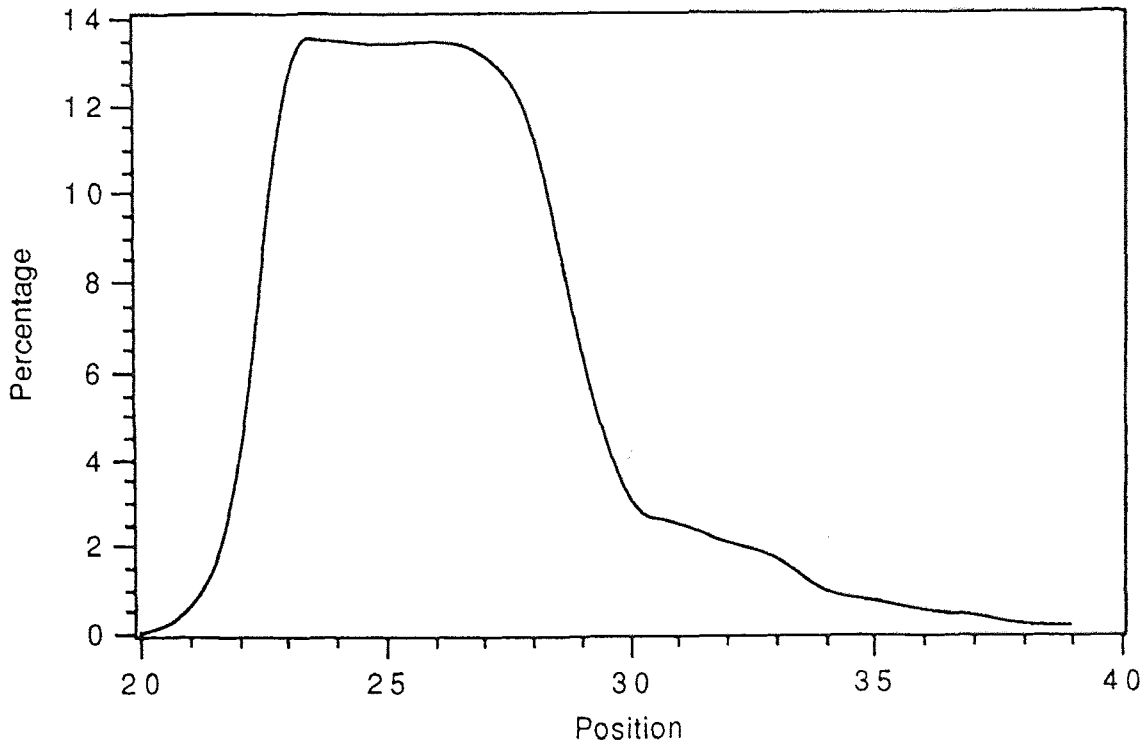

Fig. 4-13 The Relative Frequency of Correction at Each Position in the Correction Section of the Register When Using the Previous Character

This scheme is capable of correcting one error every 320 bits, since the size of the register is 320 bits. However, because with high probability errors are detected at the first marker, errors at closer patterns can be corrected. For example, if the error being corrected is at position 24 in the register, the next error can be corrected if it is just 192 bits far apart.

To compare these results with channel coding, we consider the Hamming code (255, 247,1) which is a perfect code capable of detecting one error every 255 bits with a 3.24% redundancy added. In our scheme we add redundancy of around 10% but we can correct one error and detect as many errors as exist in the register.

4.4.3 With the Average Marker Strategy

In the simulation of this strategy, the register length is chosen to be 20 characters, and the block length is 10 characters. For different text files, the ratio of file expansion is around 18.5% as it was calculated. The experimental curve in figure (4-14) shows the percentage of correction at each position of the correction section. Note that errors is detected at the first marker with a very high probability. This strategy works for high error rates since the register length is only 160 bits long.

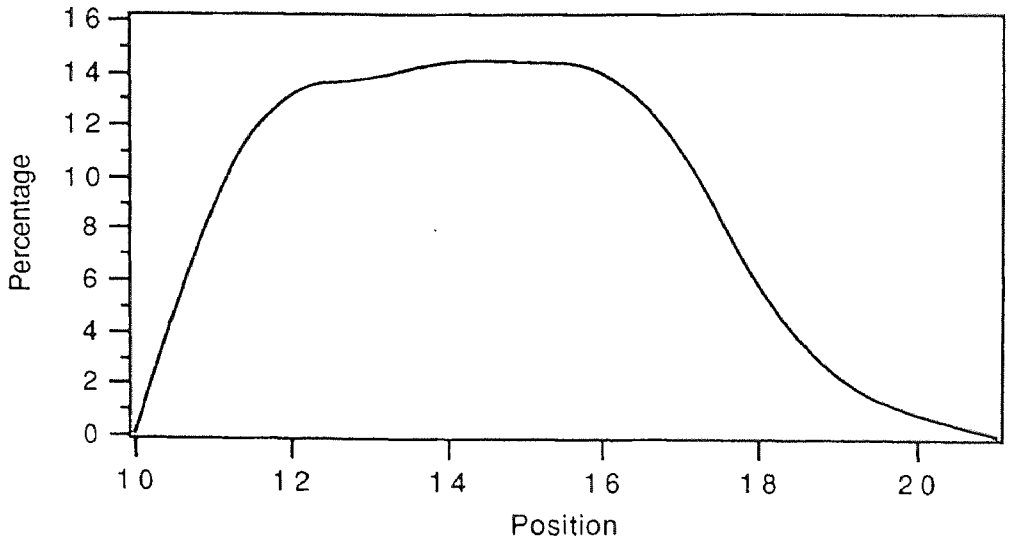

Fig. 4-14 The Relative Frequency of Correction at Each Position in the Correction Section of the Register With the Average Marker

4.5 Future Prospects

4.5.1 Towards a More Sophisticated Software

A good improvement could be achieved by using a more sophisticated software in the correction algorithm. For example, for higher error rates, when error could not be corrected because of the presence of two errors in the same block, both errors could be corrected by toggling two bits at a time. A simulation of this was done which proved that this scheme is able to correct two errors at the same time if they are both exist in the correction section of the register when we toggle. If one error is in the correction section and the other one is in the detection section, these two errors could not be corrected.

Also, in using the space character as a marker, we noticed that the misdetection and the miscorrection happened when the error is in a block that contained only space characters or most of the characters in the block were spaces. By counting the number of space characters in the block and putting a different marker (e.g. the 'e' character) when the number of space characters exceeds 10, we were able to decrease the late detection and the miscorrection considerably and hence decrease the register length.

In the next section we are going to introduce an important factor that has an effective impact on the process of detection and correction and can further improve the results.

4.5.2 The Effect of RADIX

The encoding algorithm used for compression leaves the RADIX of the arithmetic (the alphabet of the encoded data) unspecified. In the succeeded pages we will explain the effect of RADIX on the error detection and correction algorithm proposed in this chapter.

Suppose we have a code string C_k=0.0011011

This string has a decimal value C_{kd}=0.2109375

When an error happens in bit number five, C_k changes to $C_{ke} = 0.0011111$

C_{ke} has a decimal value $C_{kde} = 0.2421875$

The decimal value of the error $\Delta_1 = C_{kde} - C_{kd} = 0.03125$

When we reach bit number four in toggling; This the bit just before the erroneous bit,

C_{ke} will change to $C'_{ke} = 0.0010111$

C'_{ke} has a decimal value $C'_{kde} = 0.1796875$

The decimal value of the new error (due to toggling) is $\Delta_2 = -0.0625$

The sum of the two errors is $\Delta = \Delta_1 + \Delta_2 = -0.03125$

Notice that the values of both Δ_1 and Δ are small compared with the value of the string itself. This means that the value of the code point changes slightly even though there is an error.

Now, consider the case when C_k is short, assume $C_k = 0.101$, i.e. $C_{kd} = 0.625$

the error (in the third bit) changes it to $C_{ke} = 0.100$, i.e. $C_{kde} = 0.5$

the decimal value of the error is $\Delta_1 = C_{kde} - C_{kd} = -0.125$

toggling the second bit; which is the bit just before the erroneous bit,

C_{ke} will change to $C'_{ke} = 0.110$, and $C'_{kde} = 0.75$

The values of both Δ_1 and Δ (-0.125 and +0.125) are not small compared with the value of the string. This means that the code point value is more sensitive to errors and therefore reduces the chance of misdetection and miscorrection. We will show that Δ_1, the change in the value of the encoded string caused by the channel error, affects the misdetection and causes late detection, while $\Delta = \Delta_1 + \Delta_2$, the resulted error after toggling, affects the miscorrection.

From equation (3.17) the transformation of data string "$S_1\ S_2 \ldots\ldots\ S_k$" to the corresponding code word is given by:

$$C_k = P(s_1) + p(s_1)\,P(s_2) + p(s_1)\,p(s_2)\,P(s_3) + \cdots\cdots\cdots + p(s_1)p(s_2)\cdots\, p(s_{k-1})\,P(s_k) \qquad (4\text{-}32)$$

When an error Δ_1 is introduced to C_k, it changes to C_{ke}

$$C_{ke}= P(s_1)+ p(s_1)\, P(s_2) + p(s_1)\, p(s_2)\, P(s_2) +\cdots\cdots+ p(s_1)\, p(s_2) \cdots p(s_{k-1})\, P(s_k) + \Delta_1 \tag{4-33}$$

Assume Δ_1 is small so that C_{ke} falls in the interval $[P(s_1),P(s_1+1))$ which causes the next source character to be decoded correctly, that is, equation (3-22) becomes,

$$P(s_1)< C_{ke}< P(s_1 +1) \tag{4-34}$$

Where $P(s_1)$ is the cumulative probability of the first source character to be decoded, and $P(s_1 +1)$ is the cumulative probability of the next character.

By applying equations (3-23) and (3-24), we get the new string value (which is the new code point) $C_{ke}^{(2)}$.

$$C_{ke}^{(2)}= P(s_2) + p(s_2)\, P(s_3) + p(s_2)\, p(s_3)\, P(s_4) +\cdots\cdots+ p(s_2)\, p(s_3) \cdots p(s_{k-1})\, P(s_k) + \frac{\Delta_1}{p(s_1)} \tag{4-35}$$

From equation (4-35), to reduce the probability of misdetection, either to maximize $\frac{\Delta_1}{p(s_1)}$ or to minimize $C_{ke}^{(2)}$. It is clear that both the error Δ_1 and $p(s_1)$ (which express the code interval) is out of our control. Δ_1 is the error introduced from the channel, and $p(s_1)$ is related to the source statistics. On the other hand, $C_{ke}^{(2)}$ is controlled by the encoder and the decoder. Practically we can minimize $C_{ke}^{(2)}$ to make the value of the error larger relative to $C_{ke}^{(2)}$ by reducing the RADIX to 2, i.e. making the encoded data in binary.

Now, assume that $\frac{\Delta_1}{p(s_1)}$ is small so that, the new code string $C_{ke}^{'(2)}$ falls in the interval $[P(s_2),P(s_2+1))$, then the next character is decoded correctly which leads to a late

detection of the error. Then equation (4-35) turns to,

$$C'^{(3)}_{ke}=P(s_3)+ p(s_3)\, P(s_4)+p(s_3)\, p(s_4)\, P(s_5) +\cdots+ p(s_3)\, p(s_4) \cdots p(s_{k-1})\, P(s_k) + \frac{\Delta_1}{p(s_1)p(s_2)} \tag{4-36}$$

From equations (4-36), as more characters are decoded, the value of the last term in the equations becomes larger and hence the change in the value of the code point becomes larger. Then, the decoder starts decoding wrong characters which means that the error is propagating in the reconstructed data. This turns the late detection problem to be the misdetection problem which has been discussed in section (4-2). For more details about how the error propagate in the reconstructed data the reader is referred to [19].

Notice that equations (4-35) and (4-36) stands also for the miscorrection with the change of Δ_1 (the ~~error due~~ to the channel), to Δ (the ~~error~~ due to the channel and toggling). The late detection problem that happens in the error detection decoder, happens with the error correction decoder when we toggle a bit very close to the erronous bit. This increases the probability of miscorrection considerably.

CHAPTER 5

JOINT SOURCE AND CHANNEL CODING

Any error detection and correction algorithm considers the string of data (code word) as a vector in a vector space. By keeping each two vectors a certain distance apart, (i.e., giving each vector a certain subspace) the received vector can indicate the vector that has been sent. Error correction can then be possible. In this chapter, we will consider the [0-1) interval as a vector space, map every code word as a vector, and define the subspace reserved for it.

5.1 [0-1) Vector Space

Arithmetic encoding considers the [0-1) interval as a space and maps the string of data as a point in that interval defined by the code point and the code interval. In this scheme we will also consider the interval [0-1) as a vector space and map the encoded string as a vector defined by the code point C and the code interval W. We will find the subspace that should be reserved for each vector to enable the decoder to detect and correct a one bit error every time a decision is made.

Fig(5-1) shows how the symbol "S-1" mapped onto the interval [0-1) as a vector defined by C_1 and W_1, and how the symbol "S" is mapped as a vector defined by C_2 and W_2. W_1 is a subinterval of [0-1) and is also considered as a vector space for the next encoded symbol "S". Note that $C_2 > C_1$.

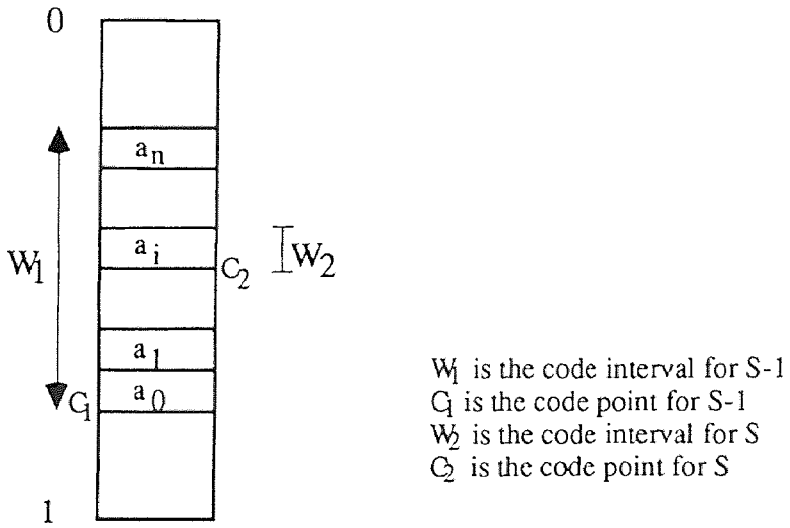

Fig. 4-1 The [0-1) Vector Space

Assume that we have a source alphabet of m symbols ($a_1, a_2, \cdots a_m$). Instead of dividing the vector space [0-1) into m subintervals as in the case of conventional arithmetic coding, we will divide it into n subintervals. $n=m * 2^L$ where L is an integer equal to (2,3 or4). The value of L depends on m and is chosen such that, the subspaces reserved for each vector of the m vectors should not overlapped, i.e. when an error happens in the vector C_2 it will changes to C'_2. C'_2 is in the subspace reserved for C_2 and indicates where the error is. In other words C'_2 is decoded as a'_i where a'_i is not one of the m symbols and points to the symbol that should be decoded.

This simply means that we do not have a source encoder and a channel encoder, but they are compressed into one stage. Also, at the decoder we do not have a channel decoder and a source decoder but just a decoder as Figure (5-2) shows.

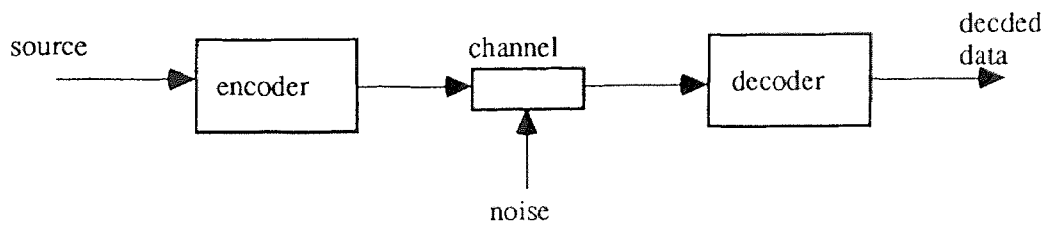

Fig. 5-2 The Communication Channel with the Suggested Scheme

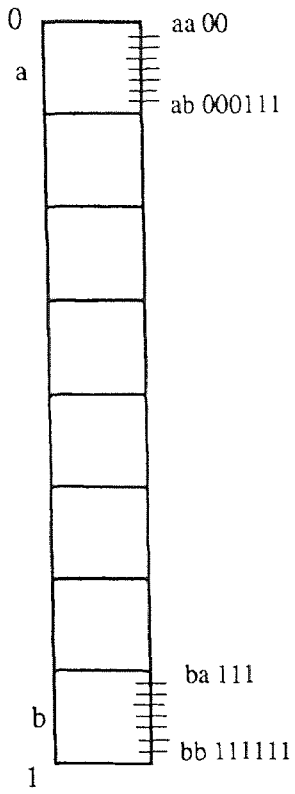

Fig. 5-3 a

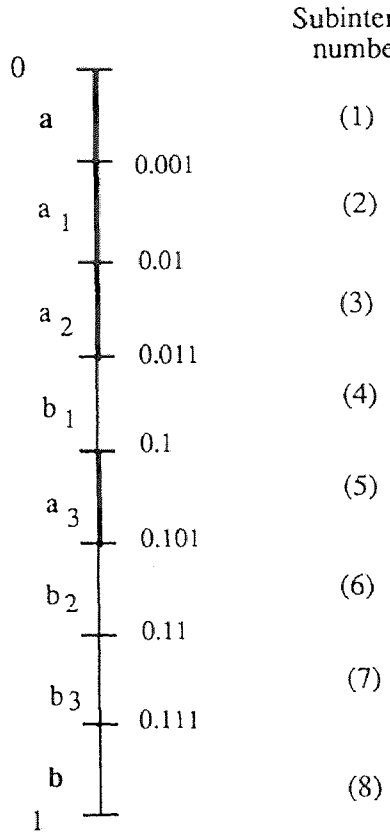

Fig. 5-3 b

Fig. 5-3 The [0-1) Vector Space for Alphabet 2

5.2 Alphabet 2

Let us consider the simplest example with alphabet = {a,b} where m=2. We choose L=2 such that n=8. This satisfies the condition that the space reserved for "a" never interferes with the space reserved for "b" as will be explained in the next paragraph. We will consider the case when the probability of (a) = the probability of (b) = 0.5, then we will generalize the case for any probability.

Figure (5-3)a shows how the vectors "a" & "b" are mapped onto the interval [0-1) to be subintervals (1) & (8) respectively. Also the figure shows how the possible sequences aa, ab ,ba and bb are mapped.

Figure (5-3)b explains how the space reserved for "a" does not overlap the space reserved for "b". Subintervals (2),(3) and (5) forms with subinterval (1) the space reserved for "a", while subintervals (4),(6) and (7) forms with subinterval (8) the space reserved for "b". Suppose the source string starts with ab, then the encoded data will start with 000111. When there is no error, the decoder first decodes "a" because 000111 falls between 0 and 001, then the first three bits are discarded (because the decoder normalizes the space for the next word to be decoded), then "b" is decoded because 111 falls in the interval reserved for "b".

Now let us apply an error to the encoded data string 000111. If the first bit is changed from 0 to 1, the encoded string will start with 100111. At the decoder, the received data string 100111 falls into subinterval (5), so it will be decoded as a_3 (see figure (5-3)b). This indicates that the transmitted data started with "a" because subinterval (5) is in the subspace reserved for "a", and hence decide that the error is in the first bit. If the error is in the second bit, the decoder will receive 010111 and decode it as a_2, which indicates that the transmitted data starts with "a" and hence decide that the error is in the second bit. Also if the error is in the third bit, the decoder will receive 001111 and decode it to a_1 which indicates that the error is in the third bit.

Now suppose that the error is in the fourth bit, the decoder will decode the string 000011 to start with "a" (which is right), then the first three bits are discarded and 011 will be decoded as b_1 which means that the next symbol to be decoded is "b" and the error is in the first bit.

Fig(5-4) shows the vector space [0-1) as the probabilities of both "a" & "b" change during arithmetic encoding. To keep the algorithm working regardless of the change in the probabilities during the process of arithmetic encoding, the probabilities of a_2, a_3 and b_3 should be equal to the probability of "a". This comes by building up the frequencies of a_2, a_3 and b_3 every time the frequency of "a" is built up. Similarly, the probabilities of a_1, b_1 and b_2 should be equal to the probability of "b". Again, this comes by building up the

frequencies of a_1, b_1 and b_2 every time the frequency of "b" is build up.

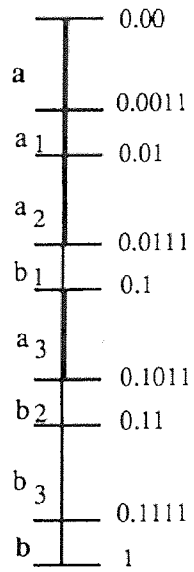

Fig. 5-4 The [0-1) Vector Space for Alphabet 2 with Unequal Probabilities

To simulate this, we built up an arithmetic encoder and decoder of source alphabet equal to eight (a, a_1, a_2, a_3, b, b_1, b_2, b_3) which give the encoded data is binary. A short file of source alphabet equal to two (a, b) were encoded such that when "a" is encoded, the frequencies of a, a_2, a_3 and b_3 are incremented. Similarly, when "b" is encoded the frequencies of b, a_1, b_1 and b_2 are incremented, and the same way for the decoder. When errors are introduced to the encoded data, the decoder receives a vector which is decoded to be any symbol but not "a" or "b". Hence the error is detected. The decoded vector tells the symbol to be decoded. On other words, the value of the received vector points out where the error is. This algorithm is able to detect and correct one error every time the decoder makes a decision.

5.3 Alphabet 4

As we have done for m=2 we will consider the case when m=4. This means that we have the source alphabet (a, b, c and d). We will choose L= 3 such that n = 32. Figure (5-5) shows the vector space [0-1) when there are equal probabilities, while Figure (5-6) shows how the vector space has been adjusted for different probabilities.

Both the encoder and the decoder do the following:

'a' is interval number 1

'b' is interval number 14

'c' is interval number 23

'd' is interval number 28

The length of intervals 5,9,13,17,21,25 and 29 is equal to the length of interval 1

The length of intervals 2,6,10,18,22,26 and 30 is equal to the length of interval 14

The length of intervals 3,7,11,15,19,27 and 31 is equal to the length of interval 23

The length of intervals 4,8,12,16,20,24 and 32 is equal to the length of interval 28

The subspace reserved for "a" is the intervals 2,3,4,5,9,17

The subspace reserved for "b" is the intervals 6,10,13,15,16,30

The subspace reserved for "c" is the intervals 7,19,21,22,24,31

The subspace reserved for "d" is the intervals 12,20,26,27,32

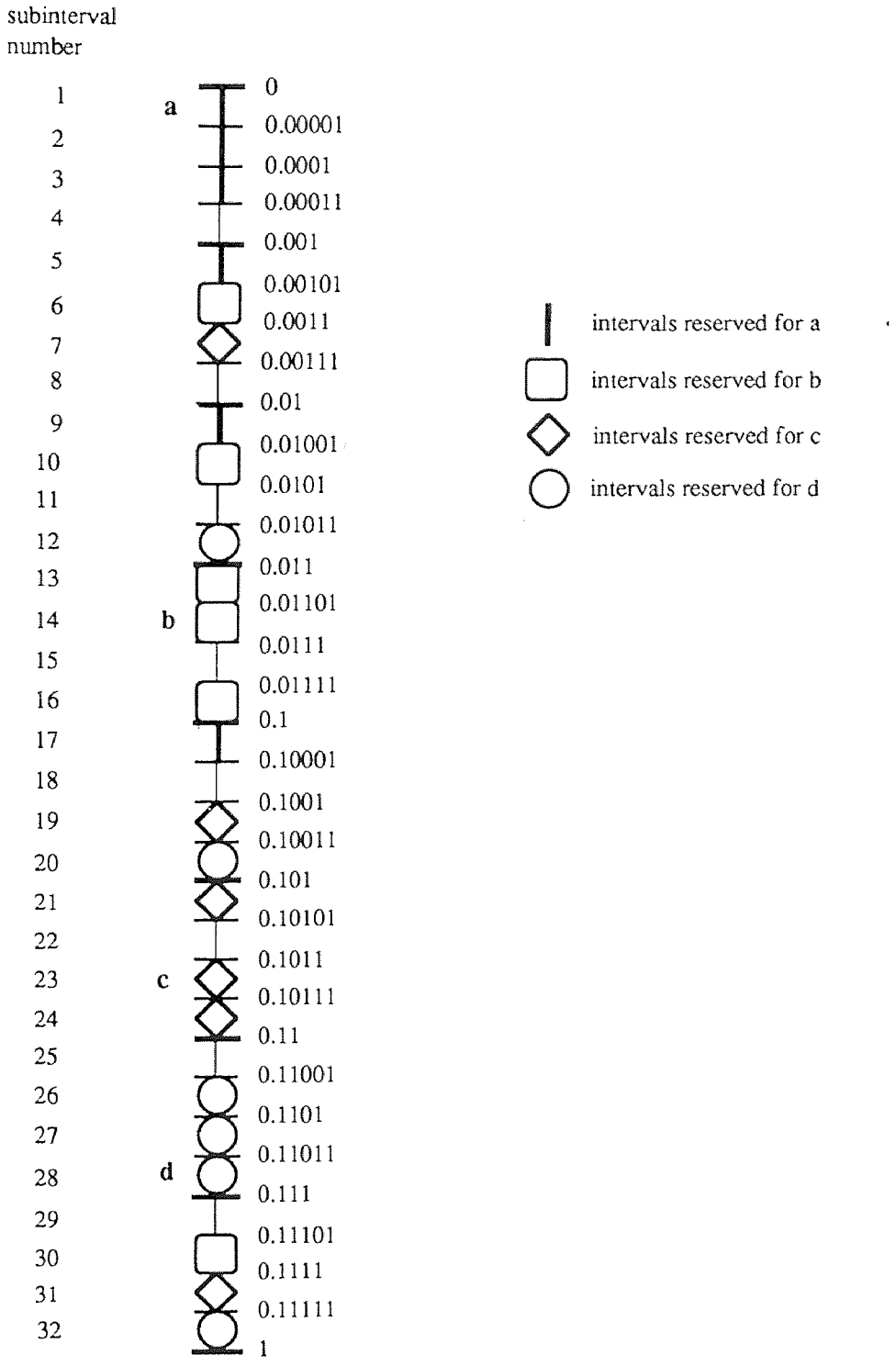

Fig. 5-5 The [0-1) Vector Space for Alphabet 4

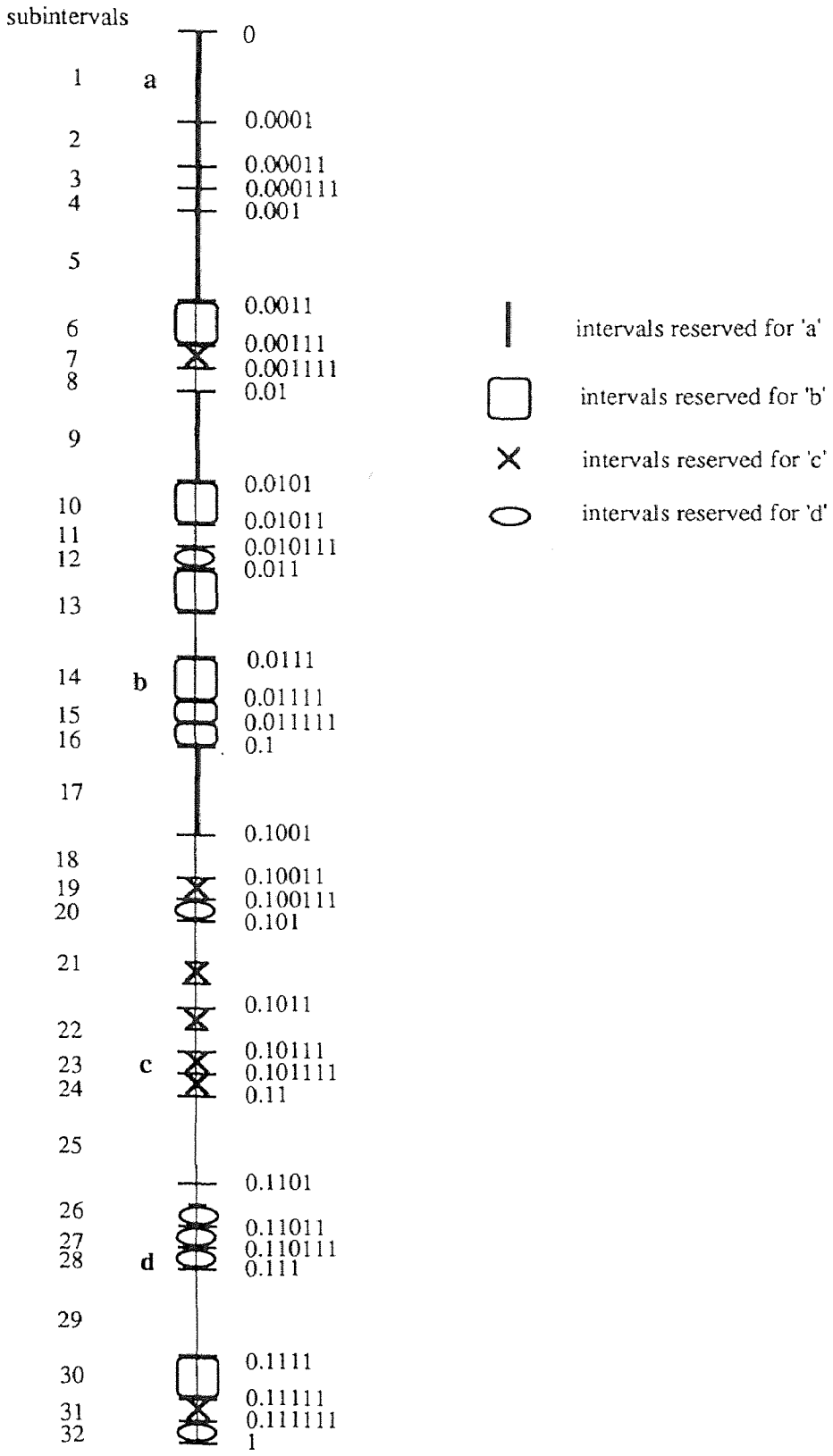

Fig. 5-6 The [0-1) Vector Space for Alphabet 4 with Unequal Probabilities

To simulate this, we built up an encoder with a source alphabet equal to 32, and encodes the data in binary. A string of an alphabet equal to 4 was encoded under the conditions explained before. When errors are introduced to the encoded data, the decoded vector indicates the symbol to be decoded, and the value of the received string points out where the error is. This system was adapted for different probabilities of the source alphabet and can correct one error every time a decision is made.

5.4 Comparison

We will compare this scheme with conventional arithmetic encoding followed by channel coding for a source alphabet equal to 4. The same string of data, will be encoded using the two methods.

For the string of data ""abacadb", Let the probability of all symbols are equal, i.e. prob(a)=prob(b)=prob(c)=prob(d)=0.25 .

Encoding this string by using conventional arithmetic encoding, as explained in the example given in section 3.3, the output of the channel encoder is the following string:

00 01 00 10 00 11 01

Assume that the channel encoder uses the block code (5,2,1) for error correction, for this string, the output of the channel encoder will be:

00000 01101 00000 10110 00000 11011 01101

Applying the proposed algorithm, the output of the encoder will be:

00000 01101 00000 10110 00000 11011 01101

which is the same string,

For both algorithms 7 errors can be corrected, one error every 5 bits. i.e. both algorithms give the same performance.

For the same string of data, let us consider that the source alphabet has different probabilities. Let prob(a)=0.5, prob(b)=0.25, prob(c)=0.125, prob(d)=0.125.

Applying conventional arithmetic encoding gives:

01 00 11 00 11 11

Applying the block code (5,2,1) we will have:

01101 00000 11011 00000 11011 11011

which is 30 bits long and can correct 6 errors, 1 error every 5 bits.

Applying the proposed algorithm we get the following string:

0000 01110 0000 101110 0000 110111 0111

which is 33 bits long and can correct 7 errors.

It is clear that we have increased the string length by 3 bits, but we can correct errors at higher rate. This is due to the fact that one error can be corrected every time the decoder takes a decision. The first four bits in the string are enough for the decoder to decide an "a". Hence, the decoder can detect and correct a one bit error in the first four bits. Because the decoder makes a decision 7 times for decoding this string, 7 errors can be corrected, while in channel coding only 6 errors can be corrected.

For the same probabilities, let us consider a different string of data "abbacaab".

Encoding this string by using conventional arithmetic encoding gives the following string:

01 01 00 11 00 01

if we use the block code (5,2,1) for this string, the output of the channel encoder will be:

01101 01101 00000 11011 00000 01101

which is 30 bits long, and one error can be corrected every 5 bits.

Applying the proposed algorithm in the previous section the encoded string will be:

0000 01110 01110 0000 101110 0000 0000 01110

which is 36 bits long, and can correct 8 errors, one error can be corrected every 4.5 bits on

the average.

To calculate the average number of bits in which the decoder can correct one error for the given probabilities (notice that the decoder uses four bits to decide "a", five bits to decide "b" and six bits to decide both "c" and "d").

$$\overline{N} = \sum_{i=0}^{n} P_i N_i \qquad (5\text{-}1)$$

where $\overline{N}$ is the average number of bits, n the number of alphabet symbols, p_i is the probability of character i, and N_i is the number of bits used to decide character i.

$$\overline{N} = 0.5 * 4 + 0.25 * 5 + 0.125 * 6 + 0.125 * 6 = 4.75 \text{ bits}$$

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," *Proceeding of the IRE*, Vol. 40, Sept. 1952, pp. 1098-1101.

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

5. T. J. Lynch, "Sequence Time Coding for Data Compression," *Proceeding of the IEEE*, Vol. 54, No. 10 1966, pp 1490-1491.

6. J. Ziv, "Coding Theorems for Individual Sequences," *IEEE Transactions of Information Theory*, IT-24 No. 3, July 1978, pp. 405-412.

7. 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.

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

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

10. Jorma Rissanen, Generalized Kraft, " Inequality and Arithmetic Coding," *IBM Journal of Reasearches and Development*, Vol. 28, No. 2, March 1984.

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

12. N. Abramson, *Information Theory and Coding*. McGraw Hill Book Company, New York, 1968.

13. Richard E. Blahut, *Principles and Practice of Information Theory*. Addison-Wesley Publishing Company,1990.

14. J. C. Lawrence, "A New Universal Coding Scheme for a Binary Memoryless source," *IEEE Transactions on Information Theory*, Vol. 23, No. 4 pp. 466-472, July 1977.

15. J. P. M. Schalwijk, "An Algorithm for Source Coding," *IEEE Transactions on Information Theory*, Vol. 18, No. 3, pp 395-399, 1972.

16. Peter Elias, "Predictive Coding," *IRE Transactions on Information Theory*, IT-1, 1955, pp. 16-44.

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

18. C. Peckham, "Word Based Data Compression Using Arithmetic Coding," Master Thesis, *Department of Electrical Engineering, New Jersey Institute of Technology*, 1988.

19. P. Narasimhan Partha, "Error Propagation and Error Correction In Universal Coding Systems," Master Thesis, *Department of Electrical Engineering, New Jersey Institute of Technology*, 1988.

20. Nadir Sizgen, "Error Detection and Correction for Compressed Data," Master Thesis, *Department of Electrical Engineering, New Jersey Institute of Technology*, 1988.

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

22. F. Jelenik, *Probabilistic Information Theory*. McGraw-Hill Book Company, New York

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

24. Shu Lin And Daniel J. Costello, Jr. *Error Control Coding*. Prentice-Hall, 1985