ADAPTIVE HOMOPHONIC CODING TECHNIQUES FOR ENHANCED E-COMMERCE SECURITY

By

David Kruger

Studyleader: Professor W.T. Penzhorn

Submitted in partial fulfillment of the requirements for the degree

Master of Engineering (Data Security)

in the

Department of Electric, Electronic and Computer Engineering

in the

Faculty of Engineering

UNIVERSITY OF PRETORIA

October 2002

SUMMARY

ADAPTIVE HOMOPHONIC CODING TECHNIQUES FOR ENHANCED E-COMMERCE SECURITY

by

David Kruger

Studyleader: Professor W.T. Penzhorn

Department of Electric, Electronic and Computer Engineering

Master of Engineering (Data Security)

This dissertation considers a method to convert an ordinary cipher system, as used to secure e-commerce transactions, into an *unconditionally secure* cipher system, i.e. one that generates ciphertext that does not contain enough statistical information to break the cipher, irrespective of how much ciphertext is available. Shannon showed that this can be achieved by maximizing the entropy of the message sequence to be encrypted. This, in turn, achieved by means of *homophonic coding.* Homophonic coding substitutes characters in the message source with randomly chosen codewords. It offers the advantage that it enables protection against *known-* and *chosen plaintext attacks* on cipher algorithms since source statistics are randomly changed before encryption. The disadvantage of homophonic substitution is that it will in general increase the length of the message sequence. To compensate for this, homophonic coding is combined with the data compression algorithm known as *arithmetic coding.* It is shown that the arithmetic coding algorithm can be adapted to perform homophonic coding by dyadically decomposing the character probabilities in its probability estimation phase. By doing this, a faster version of arithmetic coding, known as *shift-and-add* arithmetic coding can be implemented. A new method of statistical modelling, based on an Infinite Impulse Response filtering method is presented.

A method to adapt the well-known Lempel-Ziv-Welch compression algorithm to perform homophonic coding is also presented. The procedure involves a bit-wise exclusive-or randomization operation during encoding. The results show that the adapted algorithms do indeed increase the entropy of the source sequences by no more than 2 bits/symbol, and even offers compression in some cases.

Keywords:
electronic-commerce, cryptography, cryptanalysis, homophonic substitution, arithmetic coding, Lempel-Ziv-Welch compression, probability estimation.

OPSOMMING

ADAPTIVE HOMOPHONIC CODING TECHNIQUES FOR ENHANCED E-COMMERCE SECURITY

deur

David Kruger

Studieleier: Professor W.T. Penzhorn

Departement Elektries, Elektronies en Rekenaar Ingenieurswese

Meester in Ingenieurswese (Data Sekuriteit)

In hierdie verhandeling word ’n metode ondersoek wat gewone enkripsie-algoritmes, soos dié wat gebruik word om elektroniese transaksies te beskerm, te omskep in ’n *onvoorwaardelik veilige* enkripsie-stelsel, dit is, ’n stelsel wat syferteks genereer wat nie genoeg informasie bevat om die unieke ooreenstemmende skoonteks te bepaal nie, ongeag hoeveel syferteks beskikbaar is. Volgens Shannon, kan dit verkry word deur die entropie van die bronsekwensie wat geënkripteer moet word te maksimeer. Die entropie van ’n sekwensie kan gemaksimeer word deur *homofoniese kodering* daarop uit te oefen. Homofoniese kodering vervang karakters in ’n boodskapbron met willekeurig-gekose kodewoorde. Dit het ook die voordeel dat dit beskerming teen *"bekende-"* en *"gekose-skoonteks" aanvalle* op enkripsie-stelsels bied, omdat die bronstatistiek verander word voordat dit geënkripteer word. Die nadeel van homofoniese substitusie is dat dit gewoonlik die lengte van die sekwensie vermeerder. Om hiervoor te kompenseer word homofoniese kodering gekombineer met die datakompressie-algoritme wat bekend staan as *rekenkundige kodering.* Daar word getoon dat rekenkundige kodering aangepas kan word om homofoniese kodering uit te voer deur die karakterwaarskynlikhede diadies te ontbind in die waarskynlikheid-estimasie-fase. Deur dit te doen kan ’n vinniger vorm van rekenkundige kodering genaamd *skuif-en-sommeer* rekenkundige kodering gemplimenteer word. ’n Nuwe metode van waarskynlikheidestimasie, wat gebaseer is op ’n Oneindige Impuls Responsie filter metode, word voorgestel.

’n Metode om die bekende *Lempel-Ziv-Welch* kompressie-algoritme aan te pas om homofoniese kodering uit te voer word ook voorgestel. Die prosedure behels ’n bis-gewyse eksklusiewe-of ewekansig-maak operasie gedurende enkodering. Die resultate toon aan dat die aangepaste algoritmes inderdaad die entropie van bronsekwensies vermeerder met nie meer as 2 bisse/simbool, en selfs kompressie in sekere gevalle aanbied.

Sleutelwoorde:
elektroniese-handeldryf, kriptografie, kripto-analise, homofoniese substitusie, rekenkundige kodering, Lempel-Ziv-Welch kompressie, waarskynlikheidestimasie.

To my loving wife, Martie

ACKNOWLEDGEMENT

I would like to thank our Heavenly Father for giving me the ability to think and for being with me through the duration of the research.

I am grateful for prof. Penzhorn's assistance, words of comfort, support and good advice that he supplied throughout the years.

I would like to thank my family for their support and never ending concern.

And last, but certainly not least, I would like to thank Martie for being there for me and helping me get through the two years of postgraduate study.

Without the help of these people the realisation of this dissertation would not have been possible.

THANK YOU

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER ONE
INTRODUCTION

"The promise of the Internet is to be a mirror of society. Everything we want to do in the real world, we want to do on the Internet ... these things require security. Computer security is a fundamental enabling technology of the Internet; it's what transforms the Internet from an academic curiosity into a serious business tool. The limits of security are the limits of the Internet. And no business or person is without these security needs."

- Bruce Schneier, Founder and CEO of Counterpane

"Online security is becoming increasingly important to companies that intend to build their business over the Internet, especially with the sudden boom of online marketplaces. "

- Melanie Austria Farmer, Staff Writer, CNET News.com

1.1 Background

The recent growth of new communications technologies, and in particular, the Internet explosion has brought electronic commerce to the verge of widespread deployment. Nowadays it is impossible to imagine a world without Internet banking and commerce. It has become part of people's daily lives to do banking, pay bills and buy products or services online. The goal of e-commerce is to make people's lives and the tasks of businesses more comfortable and convenient. Studies have shown that the success of e-commerce sites depends largely on the trust that customers have in doing online transactions [1]. If a customer feels that a site is secure and that any private information that is sent over the Internet will be protected, the site will have a greater chance of being a success. E-commerce sites utilize the Secure Sockets Layer (SSL) protocol for secure electronic transactions. This protocol makes use of mechanisms that have been developed to secure online transactions, for instance encryption, authentication and digital certificates. It is the task of an encryption algorithm to scramble the data that represents the confidential information by making use of a secret key in such a manner that it is "impossible" to recover the original data without knowledge of the key used.

1.2 Problem Statement

In reality it is impossible to design an encryption algorithm that is unbreakable. The criteria used when designing modern encryption algorithms is usually to design an algorithm that does not make it worth the trouble to break the algorithm. Over the years methods have been developed to mount attacks on encryption algorithms [2]. For instance, by observing parts of the data sent in an electronic transaction along with its encrypted version, statistical methods can be used to discover the secret key used in the encryption process and thus compromise the security of the system. This type of attack is referred to as a *known plaintext* attack. In a *chosen plaintext* attack, the attacker chooses parts of the plaintext and observes the corresponding ciphertext. Because e-commerce transactions often involve the completion of a form with certain consistent fields, both these types of attacks are possible. This problem causes customers to loose confidence in the security offered by on-line transaction security mechanisms.

1.3 Objectives of this Study

The goal of this research is to investigate methods to design an unconditionally secure encryption algorithm. This means that no matter how much knowledge an attacker has about the information before it was encrypted, statistical analysis will not help in recovering the key used to encrypt the sequence, or equivalently, the confidential information in the sequence. It will be shown that this can be achieved by performing homophonic coding on sources before they are encrypted. The disadvantage of applying homophonic coding on source sequences is that it will in general increase the length of the sequences. To compensate for this, a method of combining homophonic coding with arithmetic coding is investigated. Arithmetic coding comprises of two distinct phases: source statistic modelling and encoding. The source modelling phase plays an important role, because the more accurately it models the true statistics of the source, the better the performance of the encoder. A new method of probability estimation, based on an Infinite Impulse Response filtering method is investigated.

Current mechanisms used to secure e-commerce will be reviewed and the theory behind enhancing security with homophonic coding will be explained. The objective is to design an adaptive homophonic coding algorithm that randomizes any given sequence, so that an unconditionally secure encryption algorithm can be designed in order to win customer trust and thus increase the success of e-commerce sites.

Even though arithmetic coding achieves much better compression than Lempel-Ziv-Welch (LZW) coding, LZW is very popular and widely used. Because of this, a method to adapt this algorithm to perform homophonic coding is also presented.

1.4 Overview of Current Literature

In 1949 Shannon wrote a seminal article that discusses the theory of secrecy systems [3]. He defines random, pure, perfect and ideal types of cryptosystems. He also defines and gives the formula for the ”unicity distance” of a random cipher.

Massey [4] realized that the criteria that Shannon proposed can be met by applying source coding to cryptography. He shows that the cascade of a binary symmetric source (BSS) and a non-expanding cipher is another BSS which yields a ciphertext sequence that is statistically independent of the secret key. According to Shannon, this is a strongly ideal cipher. Massey illustrated that a ”perfect” source coding scheme converts an information source into a BSS. This means that applying perfect source coding to an information source before encryption with a non-expanding cipher creates a strongly ideal cipher.

Günter *et al.* [5] showed that homophonic coding is the most adequate randomization technique for the purpose of constructing a strongly ideal cipher. They presented a homophonic coding algorithm that is an efficient precoding, suitable to increase the unicity distance of a cipher to any required length.

Subsequently, Witten *et al.* described how arithmetic coding, Lempel-Ziv compression and dynamic Markov modelling can be applied to enhance privacy [6]. An adaptive homophonic coding method was illustrated by Smith [7] based on the arithmetic coding implementation of Nelson [8]. Penzhorn [23] introduced a method to perform homophonic coding with the shift-and-add version of arithmetic coding, and illustrated results obtained when implementing a static probability estimation model.

1.5 Preview

The steps needed to build a trusted e-commerce infrastructure are discussed in the next chapter. The goal is to give insight into the mechanisms currently available to achieve security in e-commerce. Chapter 3 introduces the underlying concepts of homophonic coding, to illustrate the motivation for the research. In Chapter 4 the operation of the arithmetic coding algorithm is reviewed and the method of adapting it to perform homophonic coding is described. A new method of adaptive probability estimation is presented. Chapter 5 applies the same concepts for the Lempel-Ziv-Welch (LZW) compression algorithm. Chapter 6 gives simulation results for various real life sources. Chapter 7 contains the summary and conclusions of this study. Appendix A illustrates how the adapted algorithms can be implemented using the C++ programming language.

Chapter TWO
Security in E-Commerce

2.1 Introduction

The Internet has become a business tool similar in importance to the telephone network and the on-site local area network [9]. By offering products and services on the web, businesses can gain unique benefits:

- **New customers:** Anyone with an Internet connection is a potential customer. Not only are the Web storefronts open 24 hours a day, but it is also available world-wide.
- **Cost-effective delivery channel:** Many products and services can be delivered directly to customers via the Web, increasing profitability by eliminating shipping and overhead costs.
- **Streamlined enrollment:** Paper-based applications can easily be held up in the mail, and once received, have to be entered manually into computer systems, a labour intensive process that can introduce errors. By accepting applications via a secure Web site, businesses can speed application processing, reduce processing costs, and improve customer service.
- **Better marketing through better customer service:** By establishing a storefront on the Web, products and services can be customized for individual customers instead of large market segments. Businesses are able to facilitate one-to-one marketing by

capturing information about demographics, personal buying habits and preferences. By analyzing this information, enterprises can target merchandise and promotions for maximum impact, tailor web pages to specific customers and conduct effective tightly focused marketing campaigns.

With the many advantages that e-commerce offers, there are also potential risks [2]:

- **Interception:** The private content of a transaction, if unprotected, can be intercepted en route over the Internet. This is an attack on confidentiality.
- **Spoofing:** The low cost of Web site creation and ease of copying existing pages makes it easy to create illegitimate sites that appear to be published by established organizations. Credit card numbers can be obtained illegally in this manner. This is an attack on authenticity.
- **Modification:** The content of a transaction can be altered en route. This is an attack on Integrity.
- **Interruption:** A competitor or disgruntled worker might alter a Web site in order to deny potential customers its service. This is an attack on availability.

The success of an e-commerce Web site depends on the amount of trust that customers have in using the on-line facilities [1]. Customers know about these risks and have to be assured that some means of security is implemented to overcome them. There are mechanisms available today that is used to prevent these Internet attacks, and it is the purpose of the next sections to introduce these concepts. It also forms the information security background of this dissertation.

2.2 Cryptography

In order to protect against the attacks mentioned in the previous section, the following security services have to be provided by the communication system:

- **Authentication:** Customers need to be assured that they are in fact doing business and sending confidential information with an authentic entity.
- **Confidentiality:** Sensitive Internet communications and transactions, such as the transmission of credit card information, must be kept private.
- **Data Integrity:** Only authorized parties may be able to modify computer system assets and transmitted information.
- **Non-repudiation:** Neither the sender, nor receiver of a message should be able to deny the transmission.

The need for these services gave rise to the field of cryptography. *Encryption* provides some of the services directly, and others indirectly. Encryption is the process of transforming information before transmitting it to make it unintelligible to all but the intended recipient. (See Fig. 2.1). The information to be transmitted is referred to as the *plaintext.* The result of the transformation process is called *ciphertext.* Encryption employs mathematical formulas called *cryptographic algorithms,* or *ciphers,* and utilizes *keys* to encrypt or decrypt information. The basic function of an *cryptographic algorithms* is to substitute each character in the plaintext with another character

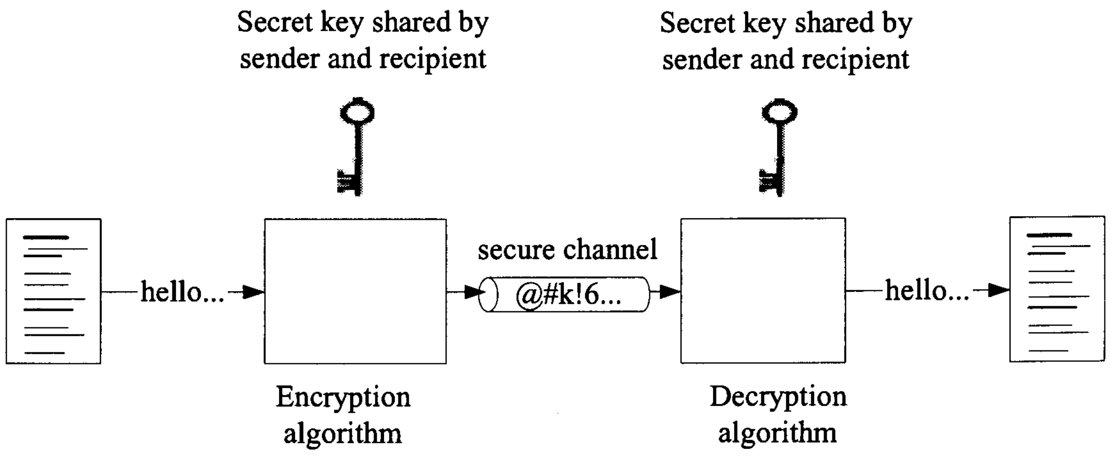

Secret key shared by sender and recipient
Secret key shared by sender and recipient
hello...
secure channel
@#k!6...
hello...
Encryption algorithm
Decryption algorithm

Figure 2.1: A Simplified Model of Symmetric Encryption

The encryption process requires a *key,* which is a secret binary number that only the sender and intended recipient know and share. Encryption algorithms are designed in such a way that it is infeasible to discover the plaintext without knowledge of the key. However, when the key is known, the plaintext can successfully be reconstructed *(decrypted)*. Encryption provides confidentiality, because no one except the receiver and the intended recipient (who are both in possession of the secret key) will be able to decrypt the data. The most well known conventional encryption algorithm is the Data Encryption Standard (DES) [2].

The following mechanisms provide the security services required for e-commerce security. The details of these mechanisms are beyond the scope of this study. (For more details, see for example any of the books on cryptography, e.g. [2], [10] or [11]:

- ***Asymmetric cryptography,*** or ***Public Key Infrastructure (PKI):*** This encryption technique has the same function as conventional encryption, but the sender and recipient do not share a secret key, and do not use the same key for encryption and decryption. The most commonly used asymmetric encryption algorithm is RSA [2]
- ***Hash functions,*** or ***Message Authentication Codes (MAC's):*** These functions make use of symmetric encryption algorithms to derive unique "fingerprints" of the data. The algorithms are designed in such a way that it is impossible to recover the original data form the "fingerprint", and no two data sequences will produce the same result. These mechanisms are used to provide data integrity, because if the sender sends the MAC with the original message, the receiver of the message can verify if the original message was tampered with en route by recreating the MAC from the received message, and comparing it with the received MAC. If the two MAC's are not the same, this indicates that the message was changed.
- ***Digital Signatures:*** Digital signatures make use of PKI and hash functions to provide a similar service as real signatures used on paper, i.e authentication.

2.3 Secure Sockets Layer

Secure Sockets Layer (SSL) is the basis for many e-commerce systems [11]. The Internet is a network of intranets that uses the Transmission Control Protocol (TCP) suite for communication. The World Wide Web (WWW) is fundamentally a client/server application running over the Internet, and because it is a public network, it is vulnerable to misuse and abuse. There are a number of approaches to providing Web security. They are similar in the services they provide, and to some extent, in the mechanisms that they use, but differ with respect to their scope of applicability and their relative location within the TCP/IP protocol stack. Security mechanisms may be applied at the network level (IPSec), the transport level (Transport Layer Security (TLS) or SSL), or the application level (e.g. S-HTTP, S/MIME, PGP and SET) [2, 9, 12]. With Internet applications, where data may be stored on several servers while in transit, it is preferable to utilize higher-level security protocols to assure reliable and secure end-to-end secure services. SSL was designed to make use of TCP for exactly this purpose [13]. SSL provides a range of security services for client server sessions:

- Server authentication
- Client Authentication
- Integrity and
- Confidentiality

The latest version, SSL v3.0, has been submitted as an Internet draft. SSL is not a single protocol, but rather consists of two layers of protocols, as illustrated in Fig. 2.2. The SSL Record Protocol is on top of the Internet TCP layer and provides basic security services to various higher layer protocols. One particular protocol that can operate on top of the SSL Record Protocol is the Hyper Text Transfer Protocol (HTTP) that is used to provide transfer services between client and server on the Web. This position of SSL between the TCP layer and application layer allows for an easy implementation on most platforms. SSL also consists of three other higher-level protocols that operate on top of the Record Protocol, namely the Handshake Protocol, the Change Cipher Spec Protocol and the Alert Protocol. The SSL

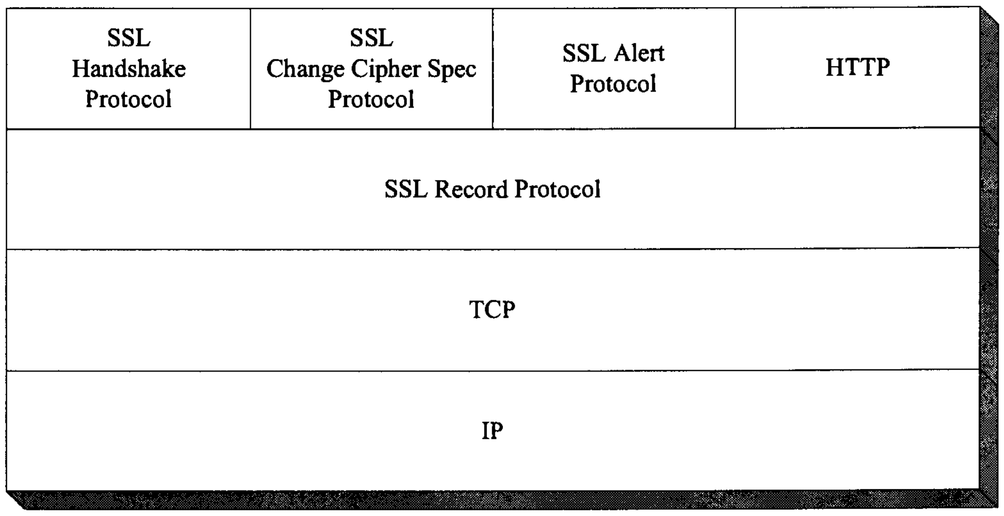


Figure 2.2: SSL Protocol Stack

Record Protocol defines the basic format for all data items sent in a session. It provides for data compression, generating an integrity check value (a MAC) on the data, and ensuring that the receiver can determine the correct data length. As part of the SSL Record Protocol, the MAC is prefixed to the data prior to encryption. To protect against the reordering of data items by an active attacker, a record sequence number is included. Cryptographic keys must first be established between client and server before the SSL Record Protocol can calculate an integrity checksum and use encryption. The protocol can also change to a different set of protection algorithms and keys at any time [12].

2.4 Cryptanalysis

Cryptanalysis is the process of attempting to recover the original plaintext, or the key that was used to encrypt it. The strategy used by the cryptanalyst depends on the nature of the encryption scheme and the information available to the cryptanalyst. Because the operation of encryption algorithms is publicly known, it is always assumed that a cryptanalyst has full knowledge of the working of the encryption algorithm. According to Stallings [2], five types of attacks are possible on encrypted messages:

- *Ciphertext only attack:* The analyst has knowledge of the encrypted ciphertext that is to be decrypted.
- *Known plaintext attack:* The analyst knows the ciphertext as well as one or more plaintext-ciphertext pairs formed with the same key.
- *Chosen plaintext attack:* The analyst knows the ciphertext and a plaintext message chosen by the analyst, together with its corresponding ciphertext generated with the secret key.
- *Chosen ciphertext attack:* The cryptanalyst knows the ciphertext and chooses purported ciphertext and observes the corresponding decrypted plaintext generated with the secret key.
- *Chosen text attack:* Again the analyst knows the ciphertext, but chooses the plaintext message and observes the corresponding ciphertext generated with the secret key. The analyst also chooses purported ciphertext and has knowledge of the corresponding decrypted plaintext generated by the secret key.

All the above mentioned attacks may be used in conjunction with a *brute-force* attack, which amounts to trying all possible keys. In most cases this is very unpractical since the key space used today is very large (e.g. 2^{56} for DES). The opponent must thus rely on an analysis of the ciphertext itself, generally applying various statistical methods and attacks. To use this approach, the opponent must have some idea of the type of plaintext that has been encrypted.

Two more definitions are noteworthy [2]. An encryption scheme is called *unconditionally secure* if the ciphertext generated by the scheme does not contain enough information to determine uniquely the corresponding plaintext, no matter how much ciphertext is available. An encryption algorithm is called *computationally secure* if the algorithm meets the following two requirements:

- The cost of breaking the cipher exceeds the value of the encrypted information.

- The time required to break the cipher exceeds the useful lifetime of the information.

The ciphertext only attack is the easiest to thwart because the opponent has the least amount of information available. In many cases, for example web-based transactions, the analyst has more information and other attacks e.g. known- and chosen plaintext attacks become possible. The analyst may be able to capture one or more plaintext messages as well as their ciphertext. Sometimes an opponent might know certain parts of the message and may only seek some very specific information in the plaintext. An example is a web page that contains a form template, where one of the fields may be a credit card number.

2.5 Discussion

The success of an e-commerce Web site depends on the measure of customer trust. The requirement is thus to implement an unconditionally secure encryption algorithm, so that a customer is ensured that the encryption algorithm can not be compromised. The next chapter explains how this can be achieved by applying source coding techniques to the encipherment process. These techniques make it possible to convert any existing encryption system into an unconditionally secure encryption system. This, in return, results in an increased customer trust to the benefit of an e-commerce Web site.

CHAPTER THREE HOMOPHONIC CODING

3.1 Introduction

This chapter explains the principle and purpose of *homophonic coding* in data security. In a sequence of symbols generated by a real-life message source, it usually happens that some symbols occur more frequently than others. In the English language, for example, the letters "e" and "t" occur more frequently in a sequence of letters than "q" and "z". Fig. 3.1 illustrates the probability of occurrence of the 26 alphabet letters in the English language, as given by [2]. The goal of homophonic coding is to convert any sequence of symbols into a

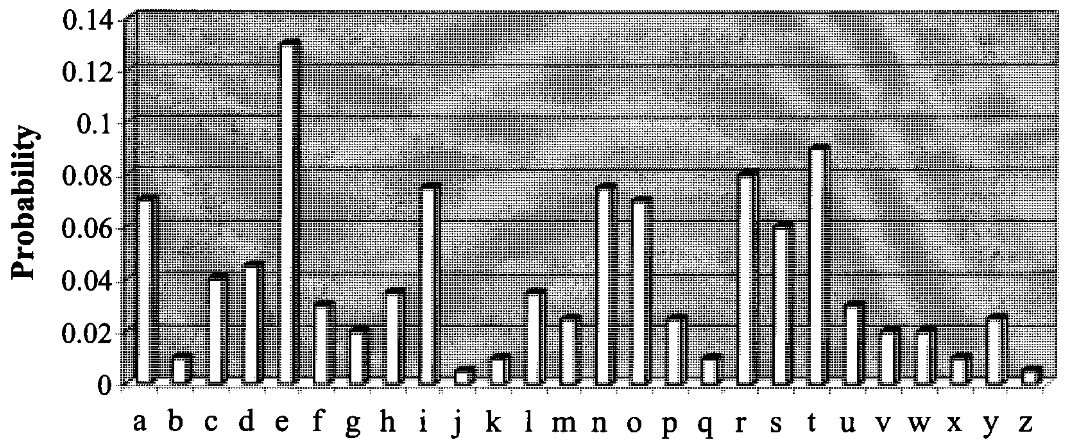


Figure 3.1: Probabilities of the alphabet letters in the English language

sequence of *equiprobable codewords,* or *homophones.* All the codewords in the resulting sequence have the same frequency of occurrence. The term *homophonic* means *to sound the same.* Homophonic substitution is a coding technique that maps source symbols onto homophones in a random fashion. Each source symbol is associated with a number of codewords chosen to represent that specific symbol, and no codeword is assigned to more than one letter. The number of homophones used to represent each source symbol is chosen to be directly proportional to the probability of occurrence of the specific symbol.

A simple example can be used to explain the concept better. Consider the English alphabet consisting of the 26 alphabet letters a-z. Suppose that the 26 alphabet letters are mapped onto the integers 1-99, where the letters with a higher relative frequency are assigned more homophones. The following illustrates a possible assignment of integers in the message *"hello there"* (for brevity, integer assignments for the remaining letters of the alphabet are not given):

$h \rightarrow 35,36,37,38$

$e \rightarrow 17,18,19,20,21,22,23,24,25,26,27,28$

$l \rightarrow 47,48,49,50$

$o \rightarrow 61,62,63,64,65,66,67,68$

$t \rightarrow 86,87,88,89,90,91,92,93,94$

$r \rightarrow 72,73,74,75,76,77,78,79,80$

One possible encoding of the message is:

Uncoded sequence:	h	e	l	l	o	t	h	e	r	e
Coded sequence:	37	27	48	50	67	91	35	17	79	22

Because there are more codewords that can represent the symbols with higher relative frequency, the homophonic substitution converts a sequence of non-uniformly distributed symbols into a sequence of uniformly distributed codewords.

3.2 Information Theory Background

3.2.1 Entropy

The entropy of a message source X that is emitting symbols $x_1, x_2, \ldots$ from an alphabet of size L is formally defined as [14]

$$H(X) = -\sum_{i=1}^{L} P(x_i) \log_2 P(x_i) \text{ bits}, \tag{3.1}$$

where $P(x_i)$ is the probability of occurrence of the ith symbol. The *entropy* of a source is the amount of information per source symbol in the language. The higher the entropy of the source, the more information (in bits) is conveyed per symbol. The entropy of a source will be maximum if all input symbols are equiprobable $(P(x_i) = 1/L \;\; \forall\, i)$[14]. Equation (3.1) then becomes $H(X) = \log_2 L$. If Y is the output of a system with an input X, the conditional entropy is defined as

$$H(X|Y) = -\sum_{i=1}^{n}\sum_{j=1}^{m} P(x_i, y_j) \log \frac{1}{P(x_i|y_j)}. \tag{3.2}$$

In this sense $H(X|Y)$ is called the *equivocation,* and is interpreted as the amount of average uncertainty remaining in X after observing Y.

3.2.2 The Rate of a Language

The *rate* of a language that consists of messages of length k, represents the *average* number of bits of information in each letter, and is defined as [10]

$$r = \frac{H(X)}{k}. \tag{3.3}$$

For English the value of r ranges from 1.0 to 1.5 bits/letter if k is large. The *absolute rate R* of a language is defined as the *maximum* number of bits of information that could be encoded, assuming that all possible sequences of letters are equally likely. If there are L letters in the language, then the absolute rate is given by

$$R = \log_2 L, \tag{3.4}$$

which is equal to the maximum entropy of the individual letters. For a 26 letter alphabet it follows that $R = \log_2 26 = 4.7$ bits/letter.

3.2.3 Redundancy

The *redundancy* of a language with rate r and absolute rate R is defined as

$$D = R - r = \log_2 L - \frac{H(X)}{k}. \tag{3.5}$$

This is a measure of redundant information in a language. The reason why compression algorithms can represent information in less bits than the uncompressed data, is because the redundancy in a language is reduced. The *percentage redundancy* is defined as

$$\rho = \frac{D}{\log_2 L} = 1 - \frac{H(X)}{k \log_2 L}. \tag{3.6}$$

3.3 Strongly Ideal and Unbreakable Ciphersystems

Jendal *et al* [15] recently published an article on an information-theoretic treatment of homophonic substitution. Some of their important results are reviewed here since they form the basis of this study.

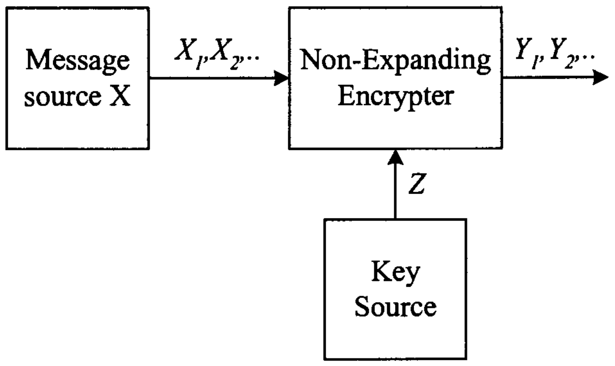


Figure 3.2: A Secret-Key Cipher System

Consider a secret key cipher system as shown in Fig. 3.2. Let a source X emit symbols $X_1, X_2, \ldots$ defined on a set X of size $L = |X|$. For simplicity of notation, let X^n and Y^n denote the (finite) plaintext and ciphertext sequences $[X_1, X_2, \ldots, X_n]$ and $[Y_1, Y_2, \ldots, Y_n]$, respectively. As is customary, and as Fig. 3.2 suggests the secret key Z is assumed to be statistically independent from the plaintext sequence X^n for all n. The system is called *non-expanding* if the plaintext symbols and ciphertext symbols take values in the same D-ary alphabet, and there is an increasing infinite sequence of positive integers $n_1, n_2, n_3, \ldots$ such that, when Z is known, X^n and Y^n uniquely determine each other for all $n \in S = \{n_1, n_2, n_3, \ldots\}$. A sequence of L-ary random variables is called *completely random* if each of its digits is statistically independent of the preceding digits and is equally likely to take on any of the D possible values.

Shannon defined the *key equivocation function,* or conditional entropy of the key, given the first n digits of ciphertext, as [3]

$$f(n) = H(Z|Y_1, Y_2, \dots Y_n) = H(Z|Y^n). \tag{3.7}$$

The key equivocation function is thus a measure of values of the secret key Z that are consistent with the first n digits of cipher text.

Because $f(n)$ can only decrease as n increases, Shannon called a cipher system *ideal* if $f(n)$ approaches a *non-zero* value as n tends toward infinity and *strongly ideal* if $f(n)$ is constant, i.e.

$$H(Z|Y^n) = H(Z) \qquad ; \forall\, n. \tag{3.8}$$

This is equivalent to the statement that the ciphertext sequence is statistically independent of the secret key. This is illustrated in Fig. 3.3.

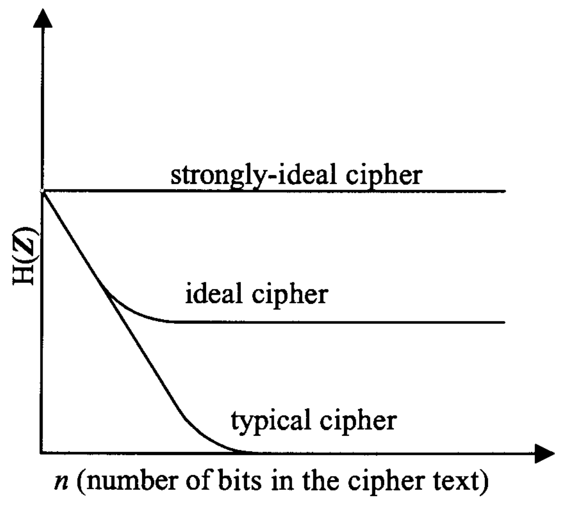


Figure 3.3: The Key-Equivocation Function

Shannon defined the *unicity distance,* n_u of a cipher as the smallest value of n such that there is essentially only one value of the secret key that is consistent with n cipher text letters $Y_1, Y_2, \dots, Y_n$ or, equivalently, such that

$$H(Z|Y_1, Y_2, \dots, Y_n) \approx 0. \tag{3.9}$$

For a specific class of cipher which Shannon termed "random cipher", it follows that

$$n_u = \frac{H(Z)}{\rho}. \tag{3.10}$$

In words, the unicity distance is the amount of ciphertext needed (in theory) to break the cipher in a ciphertext only attack. Thus, in the case of a strongly ideal cipher system the redundancy $D = 0$, so that the unicity distance tends to infinity, as illustrated in Fig. 3.3. This implies that the message source emits completely random plaintext, so that

$$H(X) = k \log_2 L. \tag{3.11}$$

Proposition 1 (Jendal *et al.*[15]):

If the plaintext sequence encrypted by a non-expanding secret key system is completely random, then the cipher sequence is also completely random, and is also independent of the secret key.

Corollary 1

If the plaintext sequence encrypted by a non-expanding secret key is completely random, then the cipher is strongly ideal (regardless of the probability distribution for the secret key).

Corollary 2

If the plaintext sequence encrypted by a non-expanding secret key is completely random and all possible key values are equally likely, then the conditional entropy of the plaintext sequence, given the ciphertext sequence, satisfies

$$H(X^n|Y^n) \approx H(Z) \tag{3.12}$$

for all n sufficiently large.

The last corollary implies that, in a ciphertext-only attack, the cryptanalyst can do no better to find X^n than by guessing at random from among as many possibilities as there are possible values for the secret key Z. In other words, the cipher system is unbreakable in a ciphertext only attack when the number of possible key values is large.

The purpose of this discussion is to illustrate that virtually any secret key cipher system can be used as the cipher in a strongly ideal cipher system, *provided that the plaintext source emits a completely random sequence*, as noted previously. But it is precisely the goal of homophonic substitution to convert a source which is not of this type into such a source. When the homophonic coding is *perfect*, it is then an easy task to build an unbreakable cipher system in the form shown in Fig. 3.4. This requires that the plaintext statistics must be known *exactly*.

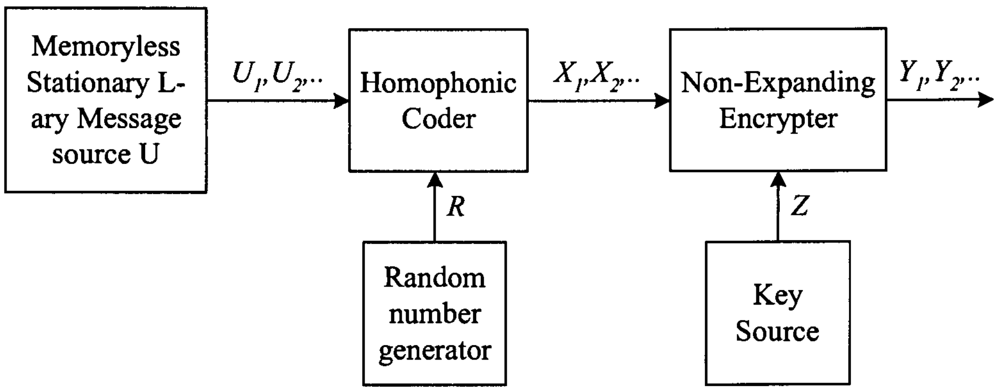


Figure 3.4: A Scheme for a Strongly Ideal Cipher System

3.4 Conventional- and Variable Length Homophonic Substitution

Consider again the message source U in Fig. 3.4, emitting symbols $U_1, U_2, \ldots$, and assume that the variables U_i take values in an alphabet of L letters where $2 \leq L < \infty$. These variables are coded into the D-ary sequence $X_1, X_2, \ldots$. Note that if $L = D^w$ for some positive integer w and all L possible values of U are equally likely, a simple coding scheme of assigning a different one of the D^w D-ary sequences of length w to each value of U will result in a completely random sequence $X_1, X_2, \ldots, X_w$. *Conventional homophonic substitution* attempts to achieve this same result when the values of U are not equally likely. This is achieved by choosing a w so that $D^w > L$ and then partitioning the D^w D-ary sequences into L subsets. These subsets are then placed in correspondence with the values of U in such a manner that the number of sequences in each subset is proportional to the probability of the corresponding value of U. A codeword is then chosen to represent a particular value u of U by an equally likely choice from the subset of sequences corresponding to u. Note that conventional homophonic substitution for which $X_1, X_2, \ldots, X_w$ is completely random is only possible if each value u_i of U has a probability n_i/D^w, where n_i is the number of homophones that must be assigned to u_i.

Fig. 3.5 gives an example of a conventional homophonic coder with $D = 2$, $w = 2$ and a binary message source ($L = 2$) taken from Jendal *et al.* [15]. From the figure it can

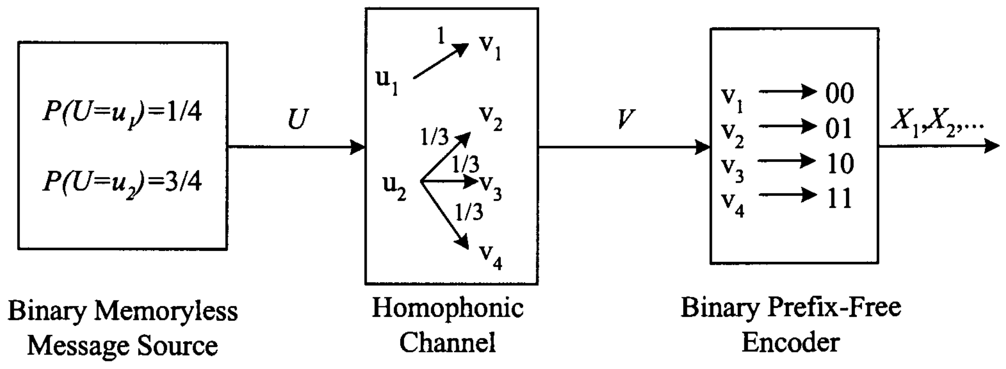


Figure 3.5: Perfect Conventional Fixed Length Homophonic Coder

be seen that the homophonic coder of Fig. 3.4 comprises of a homophonic channel and a binary prefix-free encoder. The homophonic channel is a memoryless channel whose input alphabet $u_1, u_2, \ldots, u_L$ is either finite, or countably infinite, and whose transition probabilities $P(V = v_j|U = u_i)$ have the property that for each j there is exactly one i such that $P(V = v_j|U = u_i) \neq 0$. These transition probabilities are used to govern the random choice of v_j to represent u_j. A D-ary prefix free encoder is a device that assigns a D-ary sequence to each v_j under the constraint that this codeword is neither the same as another codeword nor forms the first part (the prefix) of a longer codeword. A homophonic coding scheme is said to be *perfect* if the encoded D-ary sequence is completely random.

Variable-length homophonic substitution, introduced by Günter *et al.*[5] generalizes the conventional scheme in that the D-ary sequences used can have different lengths. If this is the case, the sequences in the subset corresponding to a given value u of U is selected with unequal probabilities in order to achieve a flat frequency distribution of symbols from the D-ary alphabet in the resulting sequence. Fig. 3.6 illustrates an example of a such a variable-length homophonic coder (also perfect), taken from Günter *et al.* [5]. When the

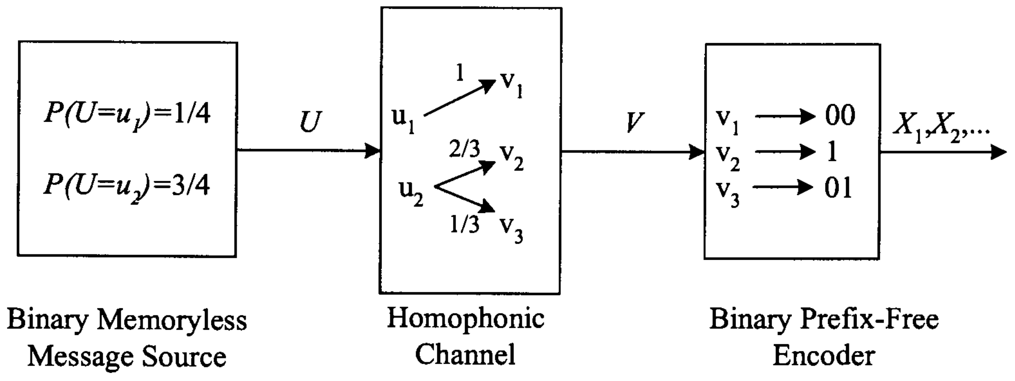


Figure 3.6: Perfect Conventional Variable Length Homophonic Coder

codeword are of unequal length, the *expected length* $E[W]$ (in bits) of a codeword can be calculated as follows:

$$E[W] = \sum_i P(v_i)l_i, \tag{3.13}$$

where l_i is the bit length of the i'th codeword and $P(v_i) = P(v_i|u_j)P(u_j)$. For the scheme in Fig. 3.6 it follows that $E[W] = \frac{1}{4}(2) + \frac{1}{2}(1) + \frac{1}{4}(2) = 1.5$ bits. On the other hand, the expected length for the codewords in the conventional homophonic substitution scheme in Fig. 3.5 is $E[W] = \frac{1}{4}(2) + \frac{1}{4}(2) + \frac{1}{4}(2) + \frac{1}{4}(2) = 2$ bits.

This illustrates the fact that variable length homophonic substitution usually results in shorter codeword sequences than conventional homophonic substitution. The source sequences in the examples have only two input symbols (i.e. a binary message source). This means that only 1 bit per symbol is needed to represent the source symbols. Because the expected length of codewords in the homophonic coding example schemes is more than 1, it can be seen that homophonic substitution results in data expansion. From Fig. 3.5 and Fig. 3.6 it is observed that the homophonic channel introduces randomness in the sequence. It is the inclusion of this additional randomness that causes the data expansion.

For memoryless sources and channels, as in Fig. 3.5 and Fig. 3.6, a homophonic substitution scheme will be perfect if the codeword $X_1, X_2, \ldots, X_w$ for $V = V_i$ is completely random [15].

Proposition 2 (Jendal *et al.*[15]):
For the homophonic schemes in Fig. 3.5 and Fig. 3.6,

$$H(U) \leq H(V) \leq E[W] \log_2 D = \sum_i P(v_i) l_i \log_2 D, \tag{3.14}$$

where $E[W]$ is the expected length of the codeword. The equality on the left holds if and only if the homophonic channel is deterministic, and the equality on the right holds if and only if the homophonic coding scheme is perfect. Also, there exists a D-ary prefix-free coding of V such that the scheme is perfect if and only if $P(V = v)$ is a negative integer power of D for all possible values v of V. When this condition is satisfied, the scheme is perfect if and only if $P(V = v_i) = D^{-l_i}$ holds for all values v_i of V, where l_i is the length of the D-ary codeword assigned to v_i. (For a proof, see [15])

From this proposition it follows that the two schemes shown in Fig. 3.5 and Fig. 3.6 are perfect, because the values of V with probability 1/4 are assigned binary codewords of length 2 and the single value of V with probability 1/2 is assigned a binary codeword of length 1.

3.5 Optimum Homophonic Substitution

A homophonic coding scheme is called *optimum* if it is perfect and minimizes the expected length $E[W]$ of the D-ary codeword.

Proposition 3 (Jendal *et al.* [15]):
A homophonic coding channel is optimum if and only if for every $u \in U$ its homophones equal (in some order) the terms in the unique dyadic[1] decomposition

$$P(U = u_i) = \sum_{j \geq 1} P(v_i)^{(j)}, \tag{3.15}$$

where the dyadic decomposition of the source probabilities is either a finite or an infinite sum.

Proposition 2 shows that the task of designing an optimum homophonic coder requires a homophonic coder that minimizes the entropy V. According to Proposition 3, this is equivalent to the requirement that the homophones are to be associated according to the dyadic decomposition of the source probabilities $P(u_i)$. It also follows from Proposition 3 that the homophonic channel in Fig. 3.6 is optimum and that $E[W] = H(V) = 3/2$ is the minimum value of $E[W]$ for perfect homophonic substitution for the message source of Fig. 3.6. Jendal *et al.* also determined the following useful upper bound on $H(V)$ for an optimum homophonic coder:

Proposition 4: ([15])
For an optimum binary homophonic coder

$$H(U) \leq H(V) = E[W] < H(U) + 2. \tag{3.16}$$

An optimum homophonic should thus never increase the entropy of its input U by more than 2 bits, regardless of how large $H(U)$ might be.

[1]The associated probability distribution of a source is called *dyadic* if the values of the symbol probabilities are distinct negative powers of two

3.6 Source Coding and Homophonic Coding

The central theme of this dissertation is to convert a practical source, having non-uniformly distributed source symbols, into a uniformly distributed message source. The main goal therefore, is to increase the unicity distance

$$n_u = \frac{H(Z)}{\rho}$$

by reducing the percentage redundancy

$$\rho = 1 - \frac{H(X)}{k \log_2 L}$$

As discussed in the previous sections, this can be accomplished by performing homophonic substitution on the source. However, reduction of redundancy can also be accomplished by data compression. The potential disadvantage of homophonic substitution is that, even though it reduces the redundancy of a source, it might also increase the total length of the source sequence, as noted by Boyd [16]. This is a direct result of the inclusion of extra randomness into the sequence, as shown in Fig. 3.4.

Data compression on the other hand reduces the redundancy and length of a source sequence, but is a deterministic mapping between the source and compressed result. This means that an attacker will not be able to launch a plaintext-only attack, but there exists a one-to-one mapping between the input source and the compressed result, so that known- and chosen-plaintext attacks would still be possible. With homophonic substitution, these two attacks are not possible, because even if the plaintext is known, or chosen, the encoded result is random (as a result of the randomly chosen homophones). In this dissertation, two methods are investigated that combine these two encoding algorithms to produce an algorithm that performs homophonic substitution, but does not increase the length of the source sequence significantly.

Note that the unicity distance (Equation 3.10) is a function of the entropy of the source $H(X)$ and the number of letters in the source alphabet L. Homophonic substitution minimizes the redundancy by maximizing the entropy, but the redundancy can be further reduced by minimizing the number of letters in the alphabet. The minimum value that L may take is 2, so it is convenient to treat the source data as a binary message source.

3.7 Discussion

The purpose of homophonic substitution is twofold: First, it denies an attacker the opportunity to launch a known- or chosen plaintext attack by randomizing the input to the encryption algorithm. Second, it maximizes the entropy of the input to the encryption algorithm, resulting in a cipher with a very large unicity distance. An attacker is thus forced to perform a ciphertext-only attack, which is essential in practice. As already mentioned, the overhead added to the sequence as a result of homophonic substitution can be removed by means of a suitable compression algorithm. There are two types of *noiseless* compression algorithms, i.e compression algorithms in which there are no information loss during compression. The first type is *fixed-to-variable* compression algorithms, in which the algorithm maps fixed length source symbols to variable length codewords, similarly to the variable length homophonic coder. The second type, *variable-to-fixed* algorithms do the opposite: it maps variable length sequences of source symbols to fixed length codewords. One of each type of source coding technique is investigated in this study: the statistical coding based fixed-to-variable *arithmetic coding* algorithm, and the dictionary based variable-to-fixed *Lempel-Ziv-Welch* data compression algorithm. These two algorithms, the reasons for choosing them and the methods to convert them into homophonic coders are discussed in the chapters to follow.

CHAPTER FOUR

HOMOPHONIC CODING BASED ON ARITHMETIC CODING

4.1 Introduction

Arithmetic coding was first introduced by Rissanen and Langdon [17, 18, 19] and may be viewed as a generalization of Shannon-Fano-Elias and Huffman coding [20]. Arithmetic coding is a compression technique that requires accurate knowledge of the source statistics. But unlike Huffman coding, it does not require that each symbol translates into a fixed code, thereby coding more efficiently. Arithmetic coding is able to achieve the theoretical entropy bound for any source [17, 19].

In arithmetic coding there exists a clear separation between modeling of source statistics and encoding of source symbols, which has distinct practical advantages. The algorithm is also easily adaptable to varying source statistics and it is not necessary to arrange symbol probabilities in any particular order, as is required for Huffman Coding. It is because of these properties that arithmetic coding was chosen for this study.

4.2 Overview of the Arithmetic Coding Algorithm

4.2.1 Compression

Arithmetic coding is a source encoding algorithm that repeatedly divides an interval into subintervals with widths proportional to the probabilities of the input symbols. The encoded data is simply the lower bound of the final interval. The process is best illustrated by means of an example.

Example:

Consider a source emitting the source symbols $\{a, b, c, d\}$ with probabilities of 1/8, 1/2, 1/4 and 1/8 respectively (see Table 4.1). The arithmetic coder is initialized with the

TABLE 4.1: **The example source statistics**

Symbol s_i	$P(s_i)$	$\sum P(s_i)$	Associated subinterval	$P(s_i)$ (in binary)	$\sum P(s_i)$	Associated subinterval
a	0.125	0	[0,0.125)	.001	.000	[.0,.001)
b	0.5	0.125	[0.125,0.625)	.1	.001	[.001,.101)
c	0.25	0.625	[0.625,0.875)	.01	.101	[.101,.110)
d	0.125	0.875	[0.875,1)	.001	.110	[.110,1)

interval [0,1), where the lower bound is closed, indicated by the square bracket [, and the upper bound is open, indicated by the round bracket). This interval is divided into four subintervals, namely [0,0.125), [0.125,0.625), [0.625,0.875) and [0.875,1), corresponding to the symbol probabilities $P(a), P(b), P(c)$, and $P(d)$ respectively, as shown in Fig. 4.1

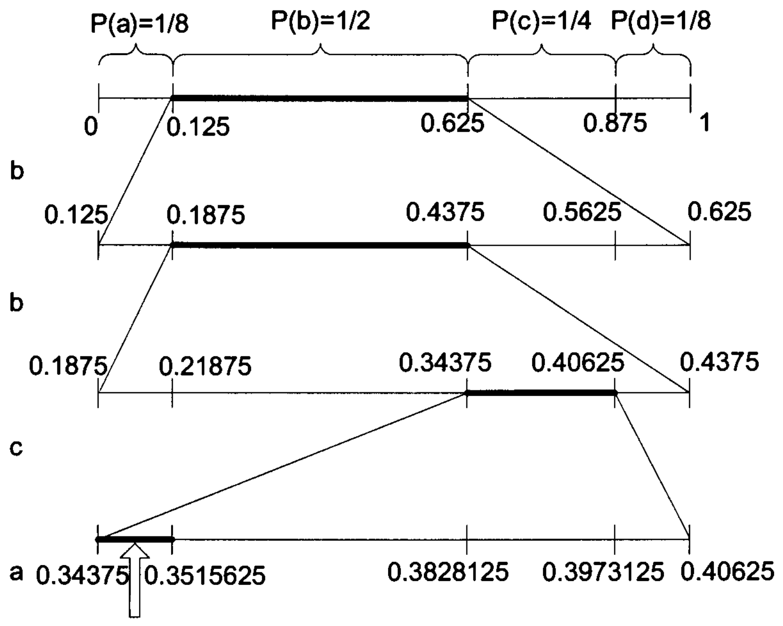


Figure 4.1: The arithmetic encoding procedure of the sequence *bbca*

Suppose that the input sequence *bbca* is to be encoded. The first step of the encoder is to choose the subinterval corresponding to the first symbol. In this case, the symbol is "*b*" and the corresponding interval is [0.125,0.625). This will be the total interval for the next iteration and is again divided into four subintervals, proportional to the symbol probabilities. The new subintervals associated with each symbol will be:

Symbol	New sub-interval width	Associated subinterval
a	$.5 \times .125 = .0625$	[.125,.1875)
b	$.5 \times .5 = .25$	[.1875,.4375)
c	$.5 \times .25 = .125$	[.4375,.5625)
d	$.5 \times .125 = .0625$	[.5625,.625)

The next input symbol to be encoded is another "*b*", and the corresponding interval is [.1875,.4375). This process is repeated until all the input symbols have been encoded, as shown in Fig. 4.1. The final subinterval is [0.34375,0.3515625) and the output of the encoder will be the lower bound, i.e. 0.34375 (or 0.01011 in binary). The process can thus be described by two recursive steps. The first step calculates the code point C, which is the lower bound of the interval corresponding to the symbol to be encoded. The second step calculates the interval width A, which is the width of the the new interval that the encoder uses for further calculations [19].

Step 1: New Code Point

The first recursion determines the new code point as the sum of the current code point C, and the product of the width A of the current interval and the cumulative probability $\sum P(s_i)$ of the symbol s_i:

$$C_k = C_{k-1} + A_{k-1} \times \sum P(s_i) \quad ; k = 1, 2, 3 \ldots \qquad \text{with } C_0 = 0 \text{ and } A_0 = 1. \tag{4.1}$$

Step 2: New interval width

The second recursion determines the width A of the new interval, which is the product of the probabilities of the data symbols encoded so far. The new interval width for the symbol s_i is

$$A_k = A_{k-1} \times P(s_i) \quad ; k = 1, 2, 3 \ldots \qquad \text{with } A_0 = 1. \tag{4.2}$$

The cumulative probabilities shown in the third column of Table 4.1 ($\sum P(s_i)$) are often referred to as *augends* [17]. When the probabilities of the source symbols are dyadic, i.e. negative powers of two, such as in this example, arithmetic coding can be reduced to the sum of augends, as shown in Table 4.2. This means that instead of making use of *dividing* and

TABLE 4.2: **Arithmetic coding as a sum of augends**

Symbol no.	Symbol	Augends						
1	b	.0	0	1				
2	b		0	0	1			
3	c			1	0	1		
4	a					0	0	0
Codeword		.0	1	0	1	1	0	0

re-scaling operations, arithmetic coding can be performed by *shifting* and *adding* operations when the source symbol probabilities are dyadic.

It should be noted that there does not exist just one *single* arithmetic coding algorithm. Rather, several classes of arithmetic coding can be identified [20, 17, 18, 19]. The *shift-and-add* method illustrated here results in a much simpler and faster arithmetic coding algorithm than any of the other cited algorithms. For example, it has been shown that the *shift-and-add* algorithm is about 25% faster than the algorithm given in [20].

4.2.2 Decompression

Decoding amounts to magnitude comparison, essentially following the inverse of the recursive steps used in the compression procedure. In order to perform decompression, the decoder must have the same information about the source statistics as the encoder to determine the interval widths. Decoding is performed in the following three steps:

Step 1: Decoder comparison

When the decoder receives 0.34375, it compares this value with the cumulative probabilities of Table 4.1. This shows that the magnitude of 0.34375 is greater than, or equal to 0.125, but less than 0.625. Hence the first received symbol is decoded as "b", since the received code string lies in the rage [0.125,0.625). Once the the symbol is decoded, it is possible to use the same recursion as the encoder to calculate the interval width:

$$A_k = A_{k-1} \times P(s_i) \;\; ; k = 1, 2, 3 \ldots \quad \text{with } A_0 = 1$$

For the first decoded symbol this gives $A_1 = 1 \times 0.5 = 0.5$.

Step 2: Decoder re-adjust

In this step the decoder subtracts the cumulative probability $\sum P(s_i)$ from the received code string. For the first decoded symbol the value $\sum P(s_2) = 0.125$ is subtracted: $.34375 - 0.125 = 0.21875$.

Step 3: Decoder scaling

During the encoding process, the the new code point C_k is determined by multiplying $\sum P(s_i)$ with the current interval width A_k. The effect of this multiplication can be "undone" by division with the interval width. This gives $0.21875/0.5 = 0.4375$.

By repeating these steps, the received sequence 0.34375 will be uniquely decoded into the encoded sequence *bbca*.

If the *shift-and-add* algorithm was used to perform compression, decompression can also be performed by "undoing" the steps in compression. In the first step it will then not be necessary to determine the interval width A_k, and the division operation in the third step becomes a left shift operation. In binary, the first symbol will be decoded as follows:

Step 1:

$0.001 \leq 0.01011 < 0.101 \therefore$ decoded symbol $= b$

Step 2:

$0.01011 - \sum P(a) = 0.01011 - 0.001 = 0.00111$

Step 3:

Left shift 0.00111 with L bits, where $P(s_i) = 2^{-L}$

$P(s_i) = P(b) = 1/2 = 2^{-1} \quad \therefore L = 1$, and the sequence becomes $0.00111 << 1 = 0.0111$, where $<< x$ means *left shift with x bits.*

By repeating these steps, the entire sequence can be decoded. The fact that the *shift-and-add* algorithm is much simpler than other algorithms, is a direct consequence of the dyadic source probabilities. At first glance the requirement that symbol probabilities are constrained to negative powers of two may appear severely restrictive. However, in Section 4.4 it will be shown how this requirement can be used to great advantage in the case of homophonic coding. The *shift-and-add* arithmetic coding algorithm is thus most suitable for the purposes of this study.

4.2.3 Implementation

4.2.3.1 Incremental Transmission and Reception

The output of the arithmetic coder is the floating point value of the lower bound of the final subinterval. In order to transmit or store this value, it must be subdivided into smaller parts that can fit into a chosen data type, for instance 16 bit unsigned integers. This means that the data will not be represented as floating point data types, but rather integers. This will allow for incremental transmission and reception.

To see this, consider again the example illustrated in the compression section (Section 4.2.1). After the first two symbols, i.e. "*bb*", are encoded, the output is 0011. Upon reception the decoder can already determine that this value lies in the range [.001,.101) and decode the first symbol as "*b*". By performing the decompression steps: 0011-0010=0001, 0001 << 1 = 0010, the second "*b*" can be decoded. Before transmitting the 16 bit integer value, care must be taken that it will not be affected by future addition operations. It is thus necessary to stall transmission until the next augend to be added is shifted far enough not to cause any changes in the most significant 16 bits.

4.2.3.2 Overflow and Underflow

Even with the above mentioned criteria an overflow can still occur. Consider the case where the transmitted integer comprises only 1's. When constructing the next integer, a bit might have to be carried over into the previous integer. Langdon [17] illustrate how the situation can be resolved with bit stuffing. This implies is that if a consecutive number of 1's occur in the stream, a 0 is inserted (*stuffed*) after the 1's. This 0 will then stop the ripple effect of a carry-over. Usually the number of 1's to be encountered before a 0 is *stuffed*, is equal to the integer word size. So if 16 bit integers are used, a 0 will be *stuffed* after encountering 16 consecutive 1's. On the decoder side the stuffed bit is removed, and if the stuffed bit is a 1, the carry is propagated inside the decoder.

Another fact that has to be taken into consideration during the compression stage, is the possibility of an underflow. This can occur because the symbol probabilities used in compression are also stored in fixed length words. The probability of a particular symbol can become too small to fit in the word. For instance, if the probability of a symbol is 2^{-17}, it can not be represented by a 16 bit integer. Bell *et al.* illustrate a method of preventing this for normal arithmetic coding [20]. This is done by limiting the maximum number of symbols to be read in at a time when probabilities are represented by the relative frequencies of characters. If no more than 2^{16} symbols are used to calculate symbol probabilities, a probability of smaller than 2^{-16} can not occur and a 16 bit integer can be used to store the probabilities.

4.3 Source Modelling

Arithmetic coding consists of two stages: the modelling of the source statistics and the actual encoding of the source symbols by utilizing the modelled statistics. This section explains the method used in this study to model the source statistics.

4.3.1 An Introduction to Source Modelling

There are a number of different issues that need attention when considering the statistical modelling of sources, often depending on the specific source in question (text, image, video etc.). Since the model for this study will be applied to arithmetic coding performing homophonic coding in e-commerce transactions, this discussion is confined to issues that apply to *text* messages. The various ways to perform statistical modelling of text sources differ in two aspects, namely the *order* of the model and whether it is *static, semi adaptive* or *adaptive* [21]. An *order-0* model is a memoryless model, which considers the probability distribution of a symbol independent of the probability distribution of any other symbol.

In an *order-n* model, the occurrence of groups of symbols of size *n* is used to determine the probability distribution. For example an *order-1* model will calculate the probability of occurrence of character pairs for example th, qu etcetera. Higher order models generally achieve better compression, but also require more memory.

The difference between static, semi-adaptive and adaptive models is the method of determining the probability of occurrence of a symbol, or a group of symbols. A static model utilizes pre-determined statistics to encode any file, and may perform very poorly [20]. A semi-adaptive model scans an entire file before encoding it to obtain the symbol probability distribution for that specific file. When such a model is utilized, the encoder must transmit the symbol probabilities to the decoder before the encoding process can begin, to make decoding possible. An adaptive model recalculates the probability distribution of the symbols on the fly, and dynamically updates the model after a certain number of symbols are encountered. An adaptive model does not scan the sequence beforehand and does not need to send the character probabilities separately. But it does require an initial model in order to operate correctly. Fig. 4.2 shows the configuration of an adaptive model. The first string of source symbols is encoded with the initial model, and then the statistics

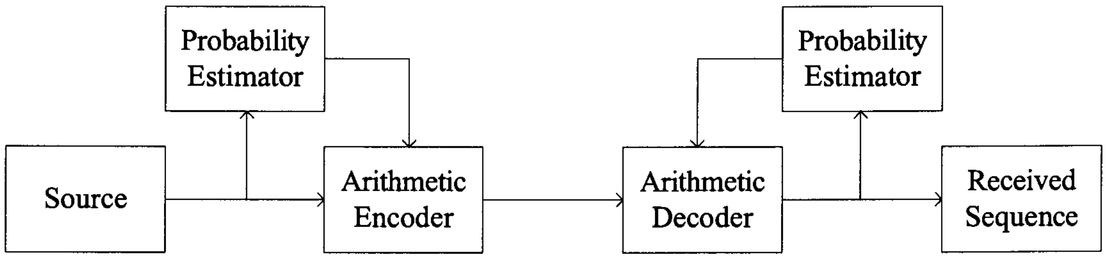


Figure 4.2: Configuration of an adaptive compression model

of this string are used to adapt the model afterwards to encode the next string of symbols. If the initial model does not accurately reflect the probabilities of the first string of symbols, which is usually the case, good compression will not be achieved initially. But after a while, the model "learns" the statistics of the file being encoded and better compression can be achieved. In general, the cost of transmitting the model when using semi-adaptive models is about the same as the "learning" cost in the adaptive case [21]. Better overall compression can however be achieved with an adaptive model because it adapts to local statistics in a file. For example, a Microsoft Word document may contain ASCII text in certain parts of the file and graphics in other parts. An adaptive model will estimate the symbol statistics more accurately at a specific point in the file than a semi-adaptive model, which uses the average statistics of the entire file to encode every part of the file. An adaptive model is more suitable for the purposes of this study.

4.3.2 The Model

The most elementary method of estimating the probability of a symbol in a sequence of symbols is to count the number of times that the symbol appears in the sequence, and to divide it by the total number of symbols in the sequence. For example, in the sequence "aabcbabcca", the symbol probabilities of the three symbols are calculated respectively as $P_a = 4/10$, $P_b = 3/10$ and $P_c = 3/10$. Over the years various methods of probability estimation have been developed, some specifically for the source that is to be modelled. For instance, it is found that in most electronic transaction data, just like in text files, the occurrence of some words or characters are clustered in some part of the file. Algorithms that take *locality of reference,* (or *recency)* [22] into account usually perfrom very well in such situations. It is thus advantageous to employ such a mechanism in the model. According to Howard *et al.* [21], there are several ways to do this in practice:

- Periodically restarting the model. This often discards too much information and is therefore ineffective.
- Using a sliding window on the text. Probabilities are calculated by counting the relative frequency of symbols in the window and by sliding the window on the text. This requires excessive computational resources.
- Recency rank coding. (Refer to [22] for a discussion). This is computationally simple but results in a rather coarse model of recency.
- Exponential aging (associating exponentially aging weights with successive symbols).

- Periodic scaling. All the symbols' counts are periodically reduced (scaled) by the same factor.

At this stage it is necessary to discuss the last two methods in more depth. According to Howard *et. al* [21] exponential aging is moderately difficult to implement because of the changing weight increments. Instead, periodic scaling is used, which is an approximation of exponential aging. The only difference is, in fact, the period at which the weights are scaled. If all weights are scaled after *every* symbol encoded, scaling and exponential aging perform *exactly* the same operations.

As for the degree of difficulty of the implementation of exponential aging, a new modelling method based on an Infinite Impulse Response (IIR) filtering technique is presented here as a simple solution. This makes it feasible to utilize the actual exponential aging algorithm, rather than an approximation of it.

The model is based on the IIR filter structure shown in Fig. 4.3. The model comprises of L such filter stages, one corresponding to each one of the L possible input symbols $s_1, s_2, \ldots, s_L$. The system processes the entire sequence of symbols one by one, i.e. adaptively. Let k denote the current symbol position in the sequence. The input to the filter corresponding to the kth symbol being encoded is a 1, while the input to all other filters equals 0. The output of each filter is scaled with a value of α, where $0 < \alpha < 1$, and is fed

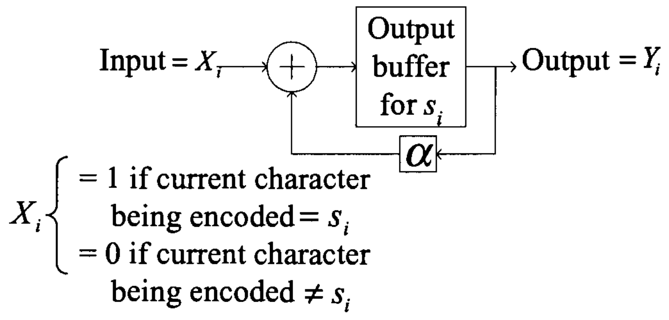

Input = X_i
+
Output buffer for s_i
Output = Y_i
α
X_i
= 1 if current character being encoded = s_i
= 0 if current character being encoded ≠ s_i

Figure 4.3: An IIR filtering scheme.

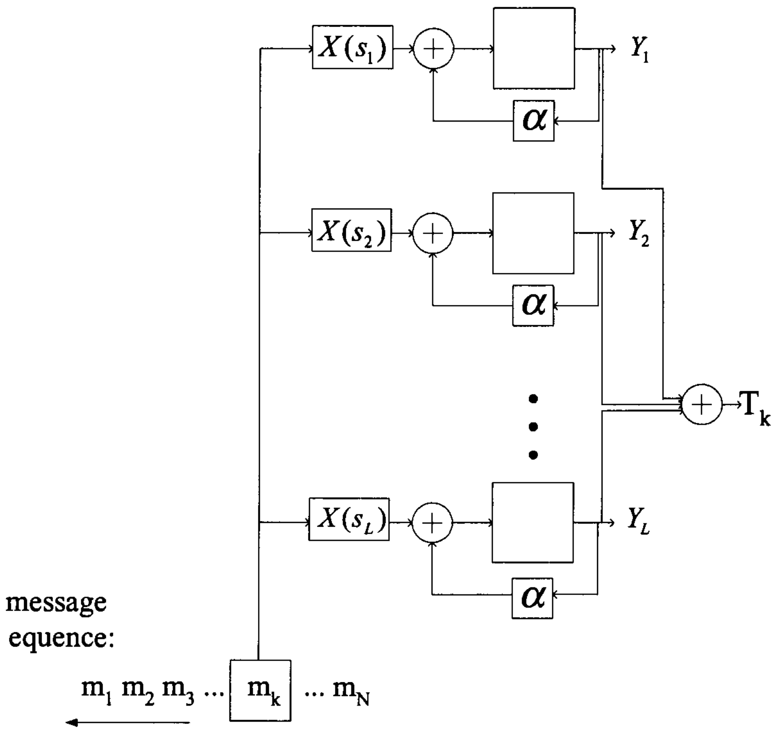


Figure 4.4: The system used for statistical modelling

back to be added to the next input value. After k symbols have been processed, the output of the system is the L probabilities associated with the L symbols in the source alphabet. Each source symbol's estimated probability $\hat{P}_i(k)$ is given by the output of its corresponding filter $Y_i(k)$, divided by the sum of all the filter outputs:

$$\hat{P}_i(k) = \frac{Y_i(k)}{\sum\limits_{j=1}^{L} Y_j(k)} = \frac{Y_i(k)}{T_k} \quad i = 1, 2, \ldots, L. \tag{4.3}$$

where T_k is defined as $T_k = \sum\limits_{i=1}^{L} Y_i(k)$. The output is thus a function of k, the number of symbols processed up to a specific point. Fig. 4.4 illustrates the entire modelling system.

4.3.3 Implementation

In arithmetic coding, a probability of 0 is not allowed. This is known as the *zero frequency problem,* thoroughly investigated by Witten and Bell [20]. A way to prevent this, is to always add a small fixed value q to the estimated probability, so that it is never 0, ensuring that all symbols will always be included in the model. The output of the modelling system is thus

$$\hat{P}_i(k) = \frac{Y_i(k) + q}{T_k + Lq} \quad i = 1, 2, \ldots, L. \tag{4.4}$$

Note that the total value of the filters' output is then equal to

$$\sum_{i=1}^{L} Y_i(k) + q = Lq + \sum_{i=1}^{L} Y_i(k) = T_k + Lq. \tag{4.5}$$

The numerator of each estimated probability thus comprises of two components: q and $Y_i(k)$. It is preferable to choose the value of q as small as possible, because it does not really contribute to the probability estimation of the symbols. (The fraction $Y_i(k)/T_k$ is the component used to estimate the symbol's probability). As mentioned in Section 4.2.3.1, the estimated probabilities are stored as integers in the computer. The smallest value that q may thus take on, is the smallest value that can be represented by the integer type it is stored in. For example, the smallest probability that can be represented by a 16-bit unsigned integer is $\frac{1}{2^{16}}$. The smallest value that $q/(T_k + Lq)$ may take on is thus $\frac{1}{2^{16}}$, giving a value of $q = \frac{T_k}{2^{16}-L}$. In general, if a register size of B bits are used, q is given the value of

$$q = \frac{T_k}{2^B - L}. \tag{4.6}$$

Consider the modelling system shown in Fig. 4.4. After the first symbol has been processed, the value of T_k is

$$T_1 = 1,$$

and after the second symbol has been processed, it is

$$T_2 = \alpha \times 1 + 1 = \alpha T_1 + 1 = \alpha + 1.$$

After the third symbol is processed, it will be

$$T_3 = \alpha \times (\alpha \times 1 + 1) + 1 = \alpha T_2 + 1 = \alpha^2 + \alpha + 1.$$

In general,

$$T_k = \alpha T_{k-1} + 1. \tag{4.7}$$

T_k is thus the k'th partial sum of the geometric series $\sum_{i=0}^{k} \alpha^i = 1 + \alpha + \alpha^2 + \ldots + \alpha^k$, which can be described as

$$T_k = \frac{1 - \alpha^{k+1}}{1 - \alpha}. \tag{4.8}$$

Since $0 < \alpha < 1$, the series converges to $\frac{1}{1-\alpha}$ when $k \to \infty$. When implementing the model, this value is stored in a floating point register of finite size, e.g as a 32-bit floating number. Because the register is of finite precision, the series will converge to $\frac{1}{1-\alpha}$ before k reaches infinity, due to the rounding of the computer. Fig. 4.5 shows different values of k at which T_k will be rounded off when stored as a 32-bit float for different values of α. For $\alpha = 0.5$, k is typically 24, which means that after 24 symbols are processed, T_k converges to 2. For $\alpha = 0.9$, $k = 179$ when T_k converges to 10, and for $\alpha = 0.99$, $k = 2113$ when T_k converges to 100.

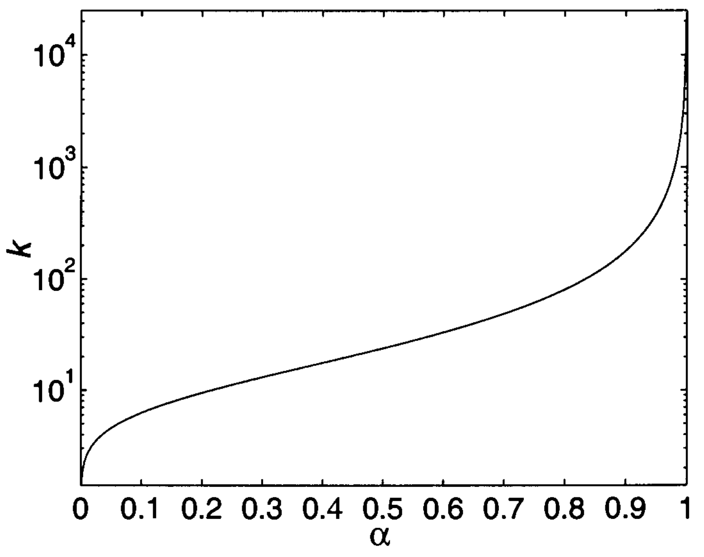


Figure 4.5: The values of k at which T_k is rounded off to its convergence value for different values of α

The purpose of this discussion is to show that it is not unreasonable to substitute T_k in Eq. 4.6 with $\frac{1}{1-\alpha}$, so that q may be calculated once as follows:

$$q = \frac{\frac{1}{1-\alpha}}{2^B - L} = \frac{1}{(1-\alpha)(2^B - L)}. \tag{4.9}$$

Because of the finite precision of the integer number that is used to store the estimated probability, the contribution of the Y_i component will reach zero after a certain number of symbols have been processed. More specifically, for a certain value of k the ratio of α^k to $T_k + Lq$ will become smaller than $1/2^B$, at which point the register entry will be rounded off to 0 (causing an underflow). The value of k at that point can be calculated by substituting Eq. 4.8 and Eq. 4.9 into

$$\frac{\alpha^k}{T_k + Lq} = \frac{1}{2^B}.$$

This gives

$$k = -\log_\alpha \left[(1-\alpha)(2^B - L) + \alpha(1 - \frac{L}{2^B}) \right]. \tag{4.10}$$

This means that after k reaches the value given in Eq. 4.10, the symbols that occurred further back than k positions have no contribution to the current estimated symbol probabilities. The graph of k for values of $0 < \alpha < 1$ is very similar to Fig. 4.5, and is illustrated in Fig. 4.6.

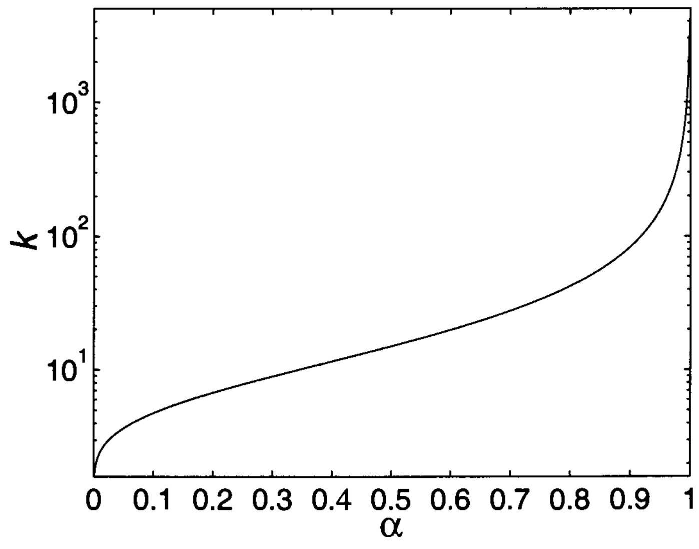


Figure 4.6: The values of k at which estimated probabilities stored in a 16-bit unsigned integer are rounded off to 0

The last point to consider in the implementation is the fact that in arithmetic coding a special end-of-file symbol has to be included in the model, to indicate to the decoder when decoding should terminate. This symbol is only coded once, at the end of the sequence, and can thus also be assigned a probability of $q/(T_k + Lq)$.

4.4 Homophonic Coding based on Arithmetic Coding

The algorithm for homophonic coding based on shift-and-add arithmetic coding comprises four steps:[23]

Step 1: Source modelling

The first step is to estimate the statistics of the source in order to determine the subinterval width associated with each source symbol. The method based on IIR filtering described in Section 4.3 is used for this purpose.

Step 2: Designing the homophonic channel

The second step is to determine the homophones $v_{i1}, v_{i2}, \ldots, v_{ij}, \ldots$ to be associated with each source symbol s_i. This is achieved by dyadically decomposing the probabilities of the source symbol. For example, if a source symbol has a probability of $P(s_1) = 1/3$, the decomposition will occur as follow:

$P(s_i)$=1/3

=0.01010101... (binary)

=0.0101 + ε (truncated)

=1/4 + 1/16 + ε

=$P(v_{i1}) + P(v_{i2}) + \varepsilon$

The two (dyadic) homophones associated with s_i are thus $v_{i1} = 0.01$ and $v_{i2} = 0.0001$. There will always occur a small error ε in the dyadic approximation as a result of the truncation of the probabilities to a finite precision. However, the magnitude of this error can be made arbitrarily small, and depends on the choice of register size. The cumulative probabilities of the homophones (the augends) represent the codewords for the homophones. Table 4.3 illustrates the process for an alphabet with non-dyadic probabilities.

TABLE 4.3: **Example of designing a homophonic channel**

Source Symbols				Homophones		
Symbol s_i	$P(s_i)$ (in decimal)	$P(s_i)$ (in binary)	$P(s_i)$ (truncated)	Symbol	$P(v_{ij})$ (in binary)	$\sum P(v_{ij})$ (in binary)
s_1 =a	.3333	.01010101	.0101	v_{11}	.0100	.0000
				v_{12}	.0001	.0100
s_2 =b	.125	.00100000	.0010	v_{21}	.0010	.0101
s_3 =c	.1666	.00101010	.0010	v_{31}	.0010	.0111
s_4 =d	.375	.01100000	.0110	v_{41}	.0100	.1001
				v_{42}	.0010	.1101

Step 3: Random selection of homophones

In the third step, each source symbol to be encoded is mapped into one of its associated homophones, chosen at random. The homophones are selected by means of an external randomiser. This introduces additional randomness into the message, which accounts for the increase in entropy of maximally 2 bits/symbol [15].

Step 4: Arithmetic coding of the homophones

The final step comprises of the encoding of each selected homophone by means of the shift-and-add arithmetic coding algorithm described in the previous section. It should be noted that any implementation of arithmetic coding could be used in this step, but the *shift-and-add* algorithm is by far the fastest [23].

4.5 Discussion

In this chapter the operation of arithmetic coding was reviewed. It was shown how source statistics can be modelled, and how arithmetic coding can be adapted to perform homophonic coding by dyadically decomposing the input symbol probabilities. By doing this, the *shift-and-add* implementation of arithmetic coding can be utilized to encode the homophones. This implementation is much faster than other implementations because it utilizes only shift and add operations.

The output of the homophonic arithmetic coder is a string of binary digits. The developed algorithm maps a given (non-uniformly distributed) sequence of source symbols into a uniformly distributed sequence of bits, and in doing so, maximizes the entropy of the sequence to be encrypted. Chapter 6 illustrates some results obtained when encoding different sources with this adapted arithmetic coding algorithm.

CHAPTER FIVE
HOMOPHONIC CODING BASED ON LZW COMPRESSION

5.1 Introduction

The Lempel-Ziv-Welch (LZW) algorithm [24] is one of the most widely used and well-known data compression algorithms. It is a *dictionary* type of data compression algorithm, that *parses* a given sequence of characters into a sequence of *substrings*, or *phrases*. The collection of these phrases is called the *dictionary*, or *codebook*. Because of its popularity, a method was investigated on how to adapt this algorithm to perform homophonic substitution as well. LZW coding is a good candidate for homophonic coding because of its *universal* properties: it automatically models any source statistics, as is needed to perform homophonic substitution.

There exist a large number of variants of the Lempel-Ziv algorithm which can all be traced back to two papers, published by Lempel and Ziv in 1977 and 1978. For a good description of most of these variants see [20]. The notations mostly used to describe these original two variants is LZ77 and LZ78. LZ77 is a "sliding window" technique in which the dictionary consists of a set of fixed-length phrases found in a "window" in the previously processed text. LZ78 takes a completely different approach to building a dictionary. Instead of using fixed-length phrases from a window into the text, LZ78 builds phrases up, one symbol at a time, adding a new symbol to an existing phrase when a match occurs. The LZ78 scheme gave rise to an article by Welch entitled "A Technique for High-Performance Data Compression" [24]. This technique was originally proposed as a method of compressing data as it is written to disk, using special hardware in the disk channel. Because of the high data rate in this application it is important that compression is very fast. This high-speed encoding is an attractive feature, especially in communication systems where encryption operations also have to be performed. The purpose of this chapter is to give an overview of the compression technique described by Welch, hereafter referred to as LZW. This chapter also illustrates the procedure used to adapt the LZW algorithm to perform homophonic substitution.

5.2 Review of the LZW Algorithm

5.2.1 Compression

In order to illustrate the operation of this algorithm, the notion of a *prefix* and a *suffix* is introduced. The suffix is sometimes also referred to as an *extension,* or *innovation character* [25]. As with any Lempel-Ziv algorithm, the object of LZW is to *parse* a given sequence of n characters into a set of distinct *phrases.* The term *phrase* is used to indicate the result of the parsing process. The last character of a phrase is the innovation character and the string of characters before it is the prefix. The collection of these phrases is called a *dictionary*, or *codebook.* With LZW the codebook is initialized by assigning all the symbols in the source alphabet to the first phrases in the dictionary. Hence, if there are L letters in the input alphabet, these will be the first L phrases in the dictionary. The LZW encoding process can then be described as follows:

1. For each phrase to be parsed off, search the codebook for a matching phrase, on a character-by-character basis. Determine the longest possible matching phrase - this is regarded as the prefix.

2. Extend the prefix by one new character, i.e. the innovation character. Place the newly

parsed phrase in the next position of the codebook, and output the codebook number that contains the prefix.

3. The next phrase begins with the innovation character of the previous phrase and the process is repeated until all characters in the input sequence have been parsed off, and their corresponding phrases placed in the codebook.

The process can be illustrated as follows. Consider a binary source ($L = 2$) emitting the following sequence of length $n = 10$:

$$1001101101$$

First, the codebook is initialized by setting entry # 0 to 0 and entry # 1 to 1. Entry # 2 is left open to serve as an End-of-File (EOF) character, to indicate when the decoding process should terminate. The first character in the sequence is a 1 and it becomes the first character of the phrase. This phrase is then extended by one character so that 10 is obtained. The sequence 10 has not yet occurred in the codebook, so it is the next phrase to be parsed off (codebook entry number 3).

TABLE 5.1: **LZW Parsing of the sequence 1001101101**

Input	Output	Dictionary entry		
		number	prefix p	IC
- $\rightarrow$	-	0	-	**0**
- $\rightarrow$	-	1	-	**1**
- $\rightarrow$	-	2	-	**eof**
10 $\rightarrow$	1	3	1	**0**
00 $\rightarrow$	0	4	0	**0**
01 $\rightarrow$	0	5	0	**1**
11 $\rightarrow$	1	6	1	**1**
101 $\rightarrow$	3	7	3	**1**
110 $\rightarrow$	6	8	6	**0**
01 $\rightarrow$	5			

10,01101101

The prefix of the phrase (1) is transmitted, and the codebook is updated. The next phrase begins with entry #3's innovation character (0). The phrase is extended by one character so that 00 is obtained. This phrase also has not yet occurred in the codebook, so it becomes codebook entry # 4, while the prefix (0) is output.

1,00,1101101

This process is repeated until all the character in the input sequences have been parsed off.

The process is summarized in Table 5.1, where the innovation characters (IC's) are shown in bold print. The output of the algorithm is therefore pointers to codebook entries. From a practical point of view it is often convenient to limit the codebook size to a fixed maximum, e.g. 2^{14}, which corresponds to a maximum codeword length of 14 bits [24]. If the codebook size used in this example was 14 bits, the result of compressing 10 bits would be 7 pointers $\times$ 14 bits = 98 bits! This illustrates the fact that with Lempel-Ziv coding, compression is usually not effective until a sizable table has been built, typically after at least 100 or so bytes have been processed [26].

5.2.2 Decompression

In the decoding process, the phrases received from the compression algorithm are used to reconstruct the input stream. One reason for the efficiency of the LZW algorithm is that it is not necessary to transmit the codebook to the receiver. At the receiver, the codebook is built exactly in the same way as during compression. First the codebook is initialized, as in the compression stage. Then each time a phrase is received, the corresponding (already existing) codebook entry is output, and used as the latest codebook entry's prefix. The first character of the next decoded phrase is the innovation character.

The compressed sequence of the example in Section 5.2.1 is decompressed as follows: The first codeword received is a 1, so 1 is output and the prefix of codebook entry #3 is 1.

The next codeword encountered is 0, so a 0 is output, codebook entry #3 is completed by appending the innovation character 0, and the prefix of codebook entry #4 is updated as 0. This is repeated until all the codewords have been decompressed, as illustrated in Table 5.2. Note that the codebook table is reconstructed *exactly* as it was in the compression stage.

TABLE 5.2: **LZW decompression of the sequence 1001365**

Input	Output	Dictionary entry		
		number	prefix p	IC
-→	-	0	-	**0**
-→	-	1	-	**1**
-→	-	2	-	**eof**
1→	1	3	1	**0**
0→	0	4	0	**0**
0→	0	5	0	**1**
1→	1	6	1	**1**
3→	10	7	3	**1**
6→	11	8	6	**0**
5→	01			

= First character of next decoded sequence

In the decoding process it can happen that a phrase is encountered that is not yet fully reconstructed in the codebook, i.e. the previous codebook entry. To see this, consider the character 1 and the string 10. If the following combination occurred in the input stream: character,string,character,string,character (1,10,1,10,1), and one of the previous codebook entries comprised of character,string (1,10) (like entry number 8 of the example), compression would occur as follows (assume codebook entry number 12 is to be constructed): parse 1101, output 8 and complete entry #12 as 1101. Next, parse 1101**x**, where **x** is the character following the last 1, output 12 and complete entry #13. When the decompression algorithm receives the codeword 8, 110 is output and the prefix of entry #12 is updated as 110. Next it receives phrase 12, but codebook entry #12 is not yet fully constructed.

The problem can however be resolved because the prefix of phrase #12 is known, so the decompressor may take its first character as the next character (innovation character), and complete the entry. The prefix of entry # 12 is entry # 8, which corresponds to 11**0**, so the first character, i.e. 1, is taken as its innovation character, and the received phrase 12 can be decoded. This is fortunately the only time that the decompressor will encounter an

TABLE 5.3: **Exception in decompression**

Input	Output	Dictionary entry		
		number	prefix p	IC
6→	11	8	6	**0**
5→	01	9	5	**⋮**
⋮→	⋮	⋮	⋮	**⋮**
8→	110	12	8	**1**
12→	1101	13	12	**x**

= First character of the prefix

undefined phrase, so that an exception handler can be added to the algorithm. The modified algorithm looks for the special case of an undefined phrase and handles it by making the first character of the uncompleted phrase its innovation character.

5.2.3 Implementation

5.2.3.1 Storing Encoding Information

In the compression stage, each time a character is encountered, a new codebook entry has to be stored. If there are, on average, 4 characters in a prefix, and if there are L letters in the input alphabet, this means storing a prefix of $5 \times \lceil log_2 L \rceil$ bits every time, counting the innovation character. ($\lceil x \rceil$ is the smallest integer larger than x). If the input to be compressed is 7-bit ASCII, an average of 35 bits have to be stored for each character processed, causing a lot of storage overhead. It also means that the algorithm has to make 35 bit comparisons when searching through the codebook each time a character processed during compression, which can make the computational overhead prohibitive. So instead of saving the entire prefix, it would be wiser to store only the codebook entry number and the innovation character. Codebook entry #8 of the example will thus be stored as **60**, instead of **110**. So for a 14-bit codebook, only 14+7=21 bits have to be stored if the input is 7 bit ASCII, or 14+1=15 bits if the input is binary ($L = 2$). It might seem that for the case of binary input, nothing is gained by storing the codebook entry number, instead of the actual prefix. But the goal of a compression is to represent a certain number of bits with less bits, so if the bit length of the prefix is more than the number of bits that it represents, no compression occurs. So the prefix length will in general exceed 14 characters. If only the codebook entry number is stored, the size of the phrase to be stored and searched will never exceed 15 bits.

5.2.3.2 Searching the Codebook

For the algorithm to be able to effectively search through the codebook, the phrases must be sorted. With a 14-bit codebook, there are potentially $2^{14} = 16384$ phrases in the codebook. Even if the phrase lengths do not exceed 14 bits, a large number of comparisons may be needed before a match is found, if it occurs at all. This problem is rectified by using a hashing algorithm to store phrases. This means that instead of storing entry #123 in position 123 of an array, it is stored in the location of the array based on the address formed by the phrase itself. When a given phrase has to be located, the phrase is used to generate a hashed address and the target phrase might be found in one search.

Nelson [26] illustrated a hashing function that, for a given phrase, bit-wise right-shifts the value of the innovation character so that its bit length equals the bit length of the prefix, and performs a bit-wise exclusive-*or* (*xor*) operation with the result and the prefix. This is used as the index of the array. If the resulting entry is unused, the phrase is stored there, and if it is in use, a comparison is made between its contents and the given phrase. If they are the same, the match has occurred and if they differ, a new index value is calculated by subtracting the current value from the table size and the process is repeated with the new index value. For this to work, the table size has to be a prime number. According to Nelson [26], the average number of searches in the table usually stays below 3 if the table size is about 25% larger than needed. For a 14 bit codebook, the array size should be a prime number higher than 1.25×2^{14}, for example 20483.

5.3 Adapting the LZW Algorithm for Homophonic Coding

When using the LZW compression algorithm to perform homophonic coding, external randomness has to be added to the compression algorithm. Penzhorn [25] introduced a randomization process for the LZ78 algorithm that performs a bit-wise *xor* operation on the entire remainder of the original sequence and a bit chosen randomly every time a phrase is parsed off. The following example illustrates the same idea for LZW:

Example:

Consider again the sequence in the example of the compression section (Section 5.2.1):

1001101101

Parsing the first phrase yields:

10,01101101

Now randomly choose a bit (say 1), and perform a bit-wise *xor* operation with the rest of the sequence:

$$\begin{array}{r} 10,01101101 \\ ,11111111 \\ \hline ,10010010 \\ \hline \end{array}$$

Now parse the next phrase:

1,01,0010010

Again, choose a random bit (say 0 this time) and perform the bit-wise *xor* operation:

1,01,0010010

,0000000

,0010010

The process is repeated until all phrases have been parsed off, as shown in Table 5.4. In this example a random bit stream of 101010 is assumed.

TABLE 5.4: **Adapted LZW Parsing of the sequence 1001101101**

Input	Output	Dict. entry			Input sequence
		#	p	IC	and *xor* operation
10 →	1	3	1	**0**	10,01101101
					11111111
01 →	0	4	0	**1**	1,01,0010010
					0000000
100 →	3	5	3	**1**	10,100,10010
					11111
00 →	0	6	0	**0**	1010,00,1101
					0000
011 →	4	7	4	**1**	10100,011,01
					11
11 →	1	8	1	**1**	1010001,11,0
					0
10 →	3				10100011,10

This randomization process effectively removes specific bit patterns that may occur in the input stream. It also decreases the redundancy of the input stream which in turn implies that less compression will be achieved by the algorithm. The resulting encoded sequence will thus generally be longer than the sequence produced by the normal LZW algorithm. In order to decode the sequence, the decoder must have knowledge of the random bits used in the encoding. This information must thus also be transmitted to the receiver, which leads to an additional overhead of 1 bit per codeword. The random bit used in the randomization procedure can be transmitted after every codeword that was transmitted. The decoder must remove this random bit from the received bit string after every codeword has been received, and use it to undo the randomization performed during encoding.

Note that this adapted algorithm does not perform homophonic coding in the traditional sense that input symbols are mapped into specific homophones. But it does produce the required properties of homophonic coding, i.e. reduces the redundancy and denies a known- or chosen plaintext attack.

In Chapter 6 simulation results are obtained for files encoded with this homophonic substitution algorithm. These results show that the sequence that the algorithm produce is not a uniformly distributed bit stream, as needed for maximum entropy. The main reason for this can be seen by observing the output of the encoder. Note that the i'th output codeword will always be less or equal than the $(i-1)$'th codebook entry. This implies that codewords transmitted before the first half of the code book is constructed will always have at least one leading 0. For example, if a 14 bit codebook is used, the first 2^{13} codewords will comprise of one or more 0's, because 14 bit words are used to represent them. Codebook entry number 9, for instance, is represented by the codeword 00000000001001. Even the 2^{13} codewords that start with a 1 obtained after the first half of the codebook has been constructed will not completely compensate for all the 0's that appear before them. An attacker can use this knowledge of 0's at specific points in the encoded bit stream to attack the encryption algorithm.

The situation can be rectified by making use of variable length codeword sizes that increase as the codebook is constructed. Because the i'th output codeword is always less or equal than the $(i-1)$'th codebook entry, only $\lceil log_2(i-1) \rceil$ bits are needed to represent that codeword. For instance, if codebook entry number 10 is being constructed, the output of the encoder can range between 0 and 9. The codeword that represents codebook entry number 9 can thus be represented with $\lceil log_2(10-1)\rceil = 4$ bits, i.e. 1001 instead of 00000000001001. With this implementation the encoder starts by sending two bits per codeword, because the first codebook entry to be formed is entry number 3, and the first codeword to be transmitted can be 0, 1 or 2. After codebook entry number 4 has been constructed, the encoder increments the number of bits to transmit per codeword to three.

In general, if the encoder is using b bits to represent the codewords at a specific point during encoding, b is incremented with one after codebook entry number 2^b has been constructed. For this implementation the decoder must follow the same pattern as the encoder. It starts by reading $b = 2$ bits at a time from the received bit string until codebook entry number 3 is being constructed. If the next received codeword is not the end-of-file codeword, the decoder continues by reading three bits at a time from then on. The decoder then increments b with one every time after entry number 2^b has been constructed. All the "unnecessary" 0's are removed in this manner, and cryptanalysts will not be able to accurately identify bit values in specific positions in the encoded bit stream. This method will from now on be referred to as the increasing-codeword-length (ICL) LZW algorithm, while the old method is referred to as the Fixed Codeword Length (FCL) LZW algorithm. The binary codewords of Table 5.4 as transmitted by the ICL LZW algorithm, follows on the next page.

TABLE 5.5: **Adapted LZW Parsing of the Sequence 1001101101 with Variable Length Codewords**

Input	Binary	Dict. entry			Input sequence
	Output	#	p	IC	and *xor* operation
10 →	01	3	1	**0**	10,01101101 11111111
01 →	00	4	0	**1**	1,01,0010010 0000000
100 →	011	5	3	**1**	10,100,10010 11111
00 →	000	6	0	**0**	1010,00,1101 0000
011 →	100	7	4	**1**	10100,011,01 11
11 →	001	8	1	**1**	1010001,11,0 0
10 →	0011				10100011,10

5.4 Discussion

In this chapter the operation of the LZW data compression algorithm was reviewed. It was shown how this algorithm can be adapted to perform homophonic coding by performing *xor* operations with the input sequence and random bits. Most implementations of the LZW algorithm use fixed length codewords to represent source symbol sequences. Some implementations however increase the codeword length as the codebook is filled up. Usually they start with a large codeword size, e.g. 12 bits [8]. In this chapter it was shown why this could be dangerous for security applications: cryptanalysts then know the bit values at certain positions in the encoded sequences. A new method that continuously increases the codeword lengths during encoding was introduced. In Chapter 6 this new technique is applied to various message sources and simulation results are given.

CHAPTER SIX
EXPERIMENTAL RESULTS

6.1 Introduction

This chapter illustrates experimental results that were obtained when encoding different source files, using the algorithms described in the previous chapters. Appendix A describes the C++ programs used to implement the algorithms. Five different source files are used in the experiments:

- A HTML file
- A C++ source file
- A TEX file
- A piece of English literature stored in an ASCII text (.txt) file
- Commercial data sent in an electronic transaction

Appendix B contains a description of the actual data contained in these files. The files are each encoded with

- a normal arithmetic encoder,
- a normal binary LZW encoder,
- the adaptive homophonic arithmetic encoder and

- the homophonic LZW encoder (FCL and ICL method).

Results are compared for the uncoded and encoded versions of the files to indicate the effectiveness of the algorithms. The lengths of the uncoded and encoded files are also compared to illustrate the effectiveness of the compression combined with homophonic coding. The following sections describe the results obtained for the C++ file. The results for the other files are shown in Appendix C (arithmetic coding results) and Appendix D LZW results). All discussions about results given in this chapter is also based on, and applicable to the results given in the appendixes.

6.2 Results of Homophonic Coding with Arithmetic Coding

The first experiment involves comparing the new IIR method of probability estimation with other methods. A convenient way to represent a set of symbol probabilities, is to calculate the entropy using Eq. 3.1. The entropy of a sequence of symbols is a single value that gives a "summary" of the probabilities. The best way to indicate the entropy at a specific point in a file is to use the sliding window method. To obtain an accurate estimate of the entropy, the rule of thumb is to utilize at least $10 \times L$ symbols to calculate the entropy, where L is the number of symbols in the source alphabet. The entropy at position k in the sequence is thus calculated utilizing the frequency of occurrence of a window of 2560 symbols preceding the kth symbol.

Fig. 6.1 illustrates the entropy of the C++ file calculated in this manner, labeled as SW Method (for Sliding Window Method). Also shown in the figure is the entropy calculated with the IIR method, labeled the IIR Method. The third plot in the figure is the method proposed by Nelson *et al.* [8], which is basically also a sliding window method. But here the window of characters used for probability estimation always ranges from the beginning of the sequence, and not just the over past 2560 symbols. It may thus be called a variable length window method (and is labeled VW Method in the figure). Recall

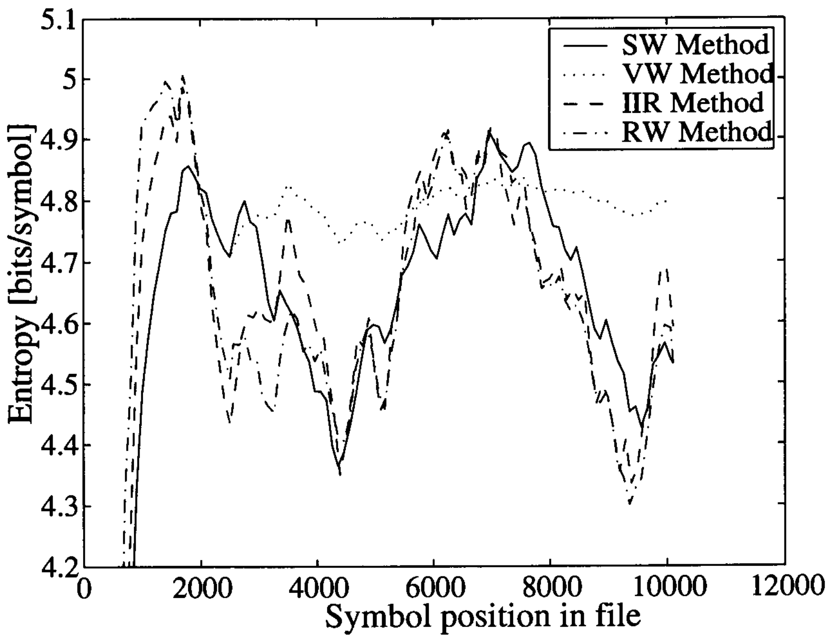


Figure 6.1: The entropy of the C++ file, as calculated by various methods.

that the IIR method is an exponential aging method. The fourth plot is a combination of a *linear* aging method and the sliding window method. It assigns linearly aging weights to successive symbols, using the past 2560 window of symbols, and disregarding any occurrences of symbols before that. This method is labeled the LA method (for Linear Aging). See Appendix H for a description of this method.

Fig. 6.1 clearly indicates that the fixed-size window methods give a better estimate of local statistics better than the variable window method, which is slow to adapt to changes in source statistics. Furthermore, it can be seen from the figure that, as more emphasis is placed on recent characters, (such as the LA method and the IIR method), the graph is "shifted" more to the left, or equivalently, the true entropy of symbols to come is being reached .

The next experiment involves comparing compression results for different values of α. Table 6.1 shows the results of encoding the C++ file with the homophonic arithmetic coding algorithm when probabilities are estimated with the IIR method. Table 6.1 shows that

TABLE 6.1: **Compression results for various values of α**

α	Length of original file	Length of encoded file	Percentage of original length
$\alpha = 0.1$	10103 bytes	14448 bytes	143%
$\alpha = 0.5$	10103 bytes	13104 bytes	129.7%
$\alpha = 0.9$	10103 bytes	9430 bytes	93.34%
$\alpha = 0.999$	10103 bytes	7280 bytes	72.06%

better compression is achieved for larger values of α. This can be explained by observing Fig. 4.6, which indicates that for small values of α, only a small number of symbols are used for probability estimation. This is due to the fact that the register size is finite, and because of round-off processes, earlier symbols are discarded. Remember that at least $10 \times 256 = 2560$ symbols are required for accurate probability estimation. According to Fig. 4.6 this happens for $\alpha \geq 0.99812$

Next, the compression results are compared for the different modelling techniques. Table 6.2 shows the length of the encoded file for each probability estimation method used. The value of $\alpha = 0.999$ was used for the IIR method. Because the length of the encoded file

TABLE 6.2: **Compression results for the C++ file for various probability estimation methods**

Method	Length of original file	Length of encoded file	Percentage of original length
SW Method	10103 bytes	6876 bytes	68.06 %
VW Method	10103 bytes	7353 bytes	72.78 %
LA Method	10103 bytes	6763 bytes	66.94 %
IIR Method	10103 bytes	6706 bytes	66.38 %

is indirectly proportional to the accuracy of the model, the results indicate that the method based on the IIR filter is the most accurate. Of all the probability estimation methods, the IIR method's exponential nature places the most emphasis on recent characters. As can be seen from the result, this proves to be of great advantage.

To illustrate the increase in entropy as a result of the adaptive shift-and-add homophonic arithmetic algorithm, the running entropy of the original file and the running entropy of the encoded file are plotted against the number of symbols encoded. The running entropy of the uncoded file is calculated by the sliding window method, while that of the encoded file is calculated by making use of Equation 3.1 with the probabilities of the homophones assigned to each interval during the encoding process. The file is encoded with the value of $\alpha = 0.999$. Fig. 6.2 shows the plot for the C++ source file.

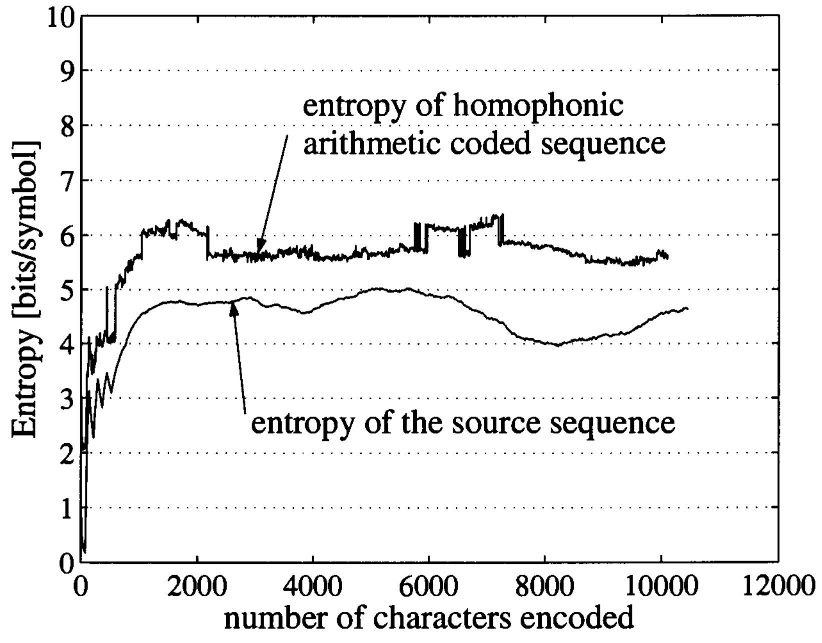


Figure 6.2: Entropy of the source and homophonic encoded C++ file

The next experiment obtains the statistics of a source file before and after it has been encoded to illustrate the effect of homophonic coding on the entropy of the sequence. Table 6.3 show the results, and Fig. 6.3 illustrates the same results graphically, for easier interpretation. Once again, the value of $\alpha = 0.999$ is used. The probabilities shown in the table are simply the relative frequency of occurrence of single bits and bit pairs, i.e. the number of times it occurs in the file divided by the total length of the file.

TABLE 6.3: **Relative frequency of occurrence of single bits and bit pairs in the uncoded and homophonic arithmetic encoded C++ source file**

Probabilities	P(0)	P(1)	P(00)	P(01)	P(10)	P(11)
Uncoded Data	0.6122	0.3878	0.3733	0.2389	0.2389	0.1489
Homophonic arithmetic encoded data	0.5025	0.4975	0.2525	0.2500	0.2500	0.2475

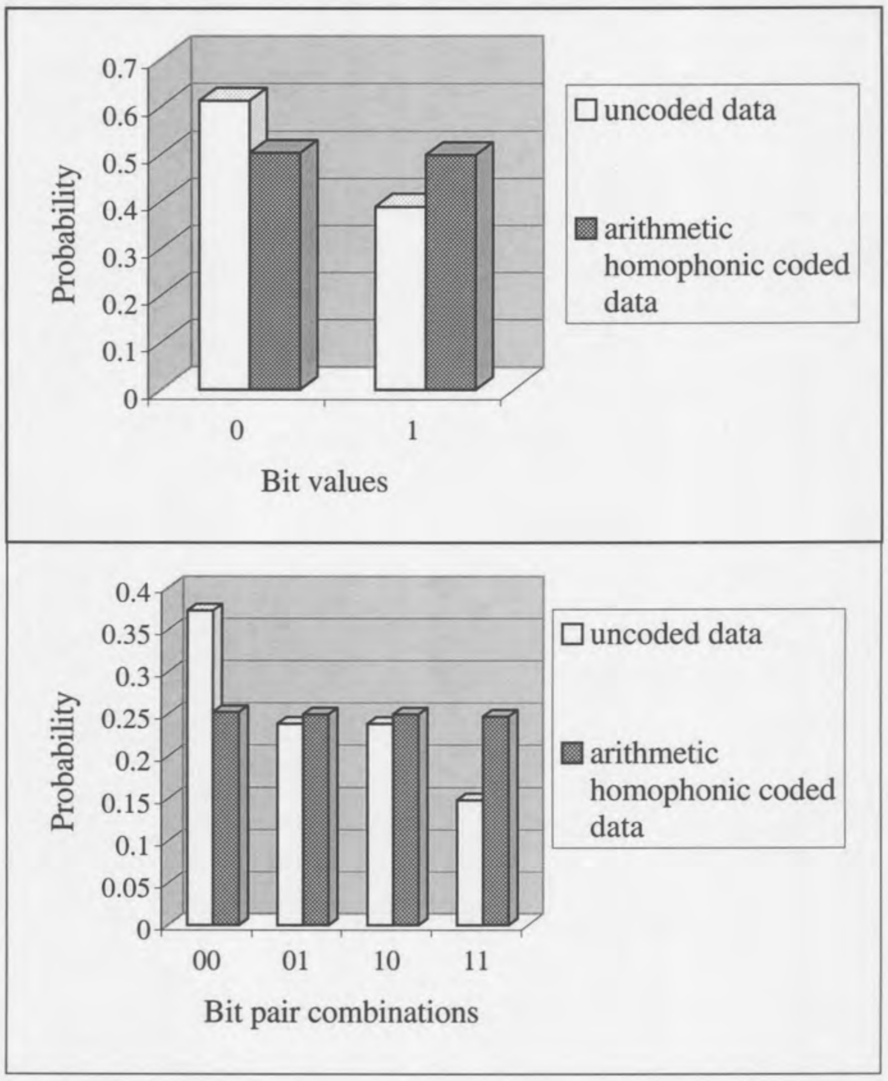


Figure 6.3: Statistics of the uncoded and encoded C++ file

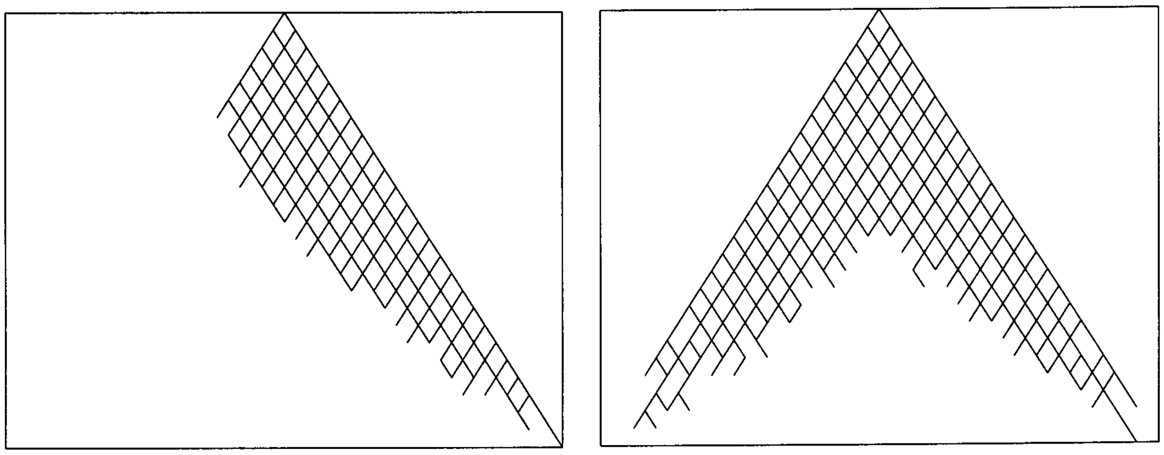
The results show that the encoded file has a nearly flat frequency distribution of bits and bit pairs, once again indicating the increase in entropy, and thus the increase of unicity distance of the encryption algorithm that is used to encrypt such files for secure transmission.

6.3 Results of Homophonic Coding with LZW Encoding

6.3.1 Bit Patterns

The first experiment illustrates the effect of homophonic coding on codebook entries formed during the LZW encoding process. These entries show certain bit patterns that occur in a source file that may lead to statistical analysis of a cipher algorithm. The goal of the homophonic algorithm is to eliminate these patterns in order to prevent successful statistical attacks on an encryption algorithm (used in known and chosen plaintext attacks). The input consists of a bit stream with $P(1) \approx 0.75$ and $P(0) \approx 0.25$. A binary tree is useful to visually illustrate bit patterns that occur in a bit stream, as shown in Fig. 6.4 (a). This tree shows a line to the right if a 1 is encountered in a codebook entry and a line to the left if a 0 is encountered. From Fig. 6.4 (a) it can be seen that the tree is biased and only certain branches of the tree are reached, as may be the case for real sources.

The goal of the homophonic coding algorithm is to randomize the input so that virtually all the branches are reached. This requires that the whole tree is filled up in a balanced way,



(a) (b)

Figure 6.4: Binary tree obtained when a bit stream with P(1)=0.75 is encoded with (a) the original binary LZW algorithm, and (b) the adapted binary LZW algorithm

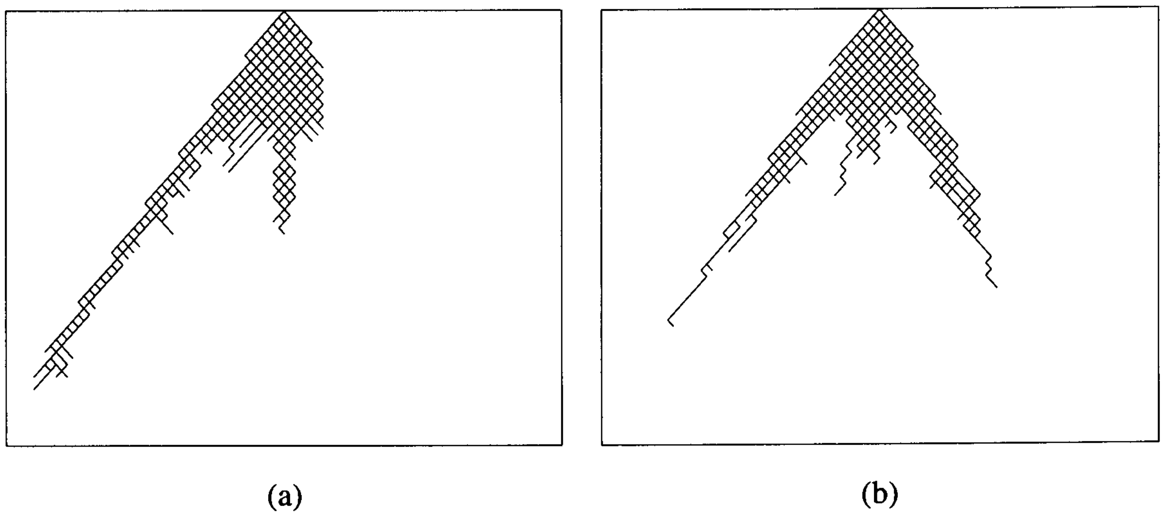


Figure 6.5: **Binary LZW trees of dictionary entries formed when encoding the C++ file with (a) normal binary LZW, and (b) homophonic LZW**

from left to right, thus indicating that the bit patterns used in statistical attacks are removed. Fig. 6.4 (b) shows the resulting codebook entries when the same bit stream is encoded with the homophonic LZW algorithm. The tree now has two main "legs", one to the right and one to the left as a result of the *xor* operation during compression. More branches are thus reached using the homophonic algorithm.

Fig. 6.5 (a) shows the tree that is formed when compressing the C++ file with the normal binary LZW algorithm and Fig. 6.5 (b) shows the tree for the same file encoded with the homophonic LZW algorithm. Note that in these figures the codebook entries are overall shorter for the homophonic coded file than for the original compressed file, indicating the removal of specific bit patterns, as needed.

6.3.2 Statistical Results

The next experiment obtains the statistics of a source file before and after it has been encoded to illustrate the effect of homophonic coding on the entropy of the sequence. Both the fixed-codeword-length (FCL) and increasing-codeword-length (ICL) LZW methods were used for this purpose. Table 6.4 contains the probabilities of occurrence of single bits and bit pairs for

- the uncoded file,
- the file compressed with binary FCL and ICL LZW,
- the file encoded with the homophonic FCL LZW algorithm and
- the file encoded with the homophonic ILC LZW algorithm.

Fig. 6.6 gives a graphical illustration of the same data.

TABLE 6.4: **Relative frequency of occurrence of single bits and bit pairs in the uncoded and encoded C++ source file**

Probabilities	P(0)	P(1)	P(00)	P(01)	P(10)	P(11)
Uncoded Data	0.6122	0.3879	0.3733	0.2389	0.2389	0.1489
FLC LZW compressed data	0.4136	0.5864	0.3532	0.2335	0.2332	0.1804
ILC LZW compressed data	0.4994	0.5006	0.2411	0.2584	0.2584	0.2422
FLC Homophonic coded data	0.5851	0.4149	0.3493	0.2359	0.2359	0.1790
ILC Homophonic coded data	0.5006	0.4994	0.2403	0.2602	0.2603	0.2392

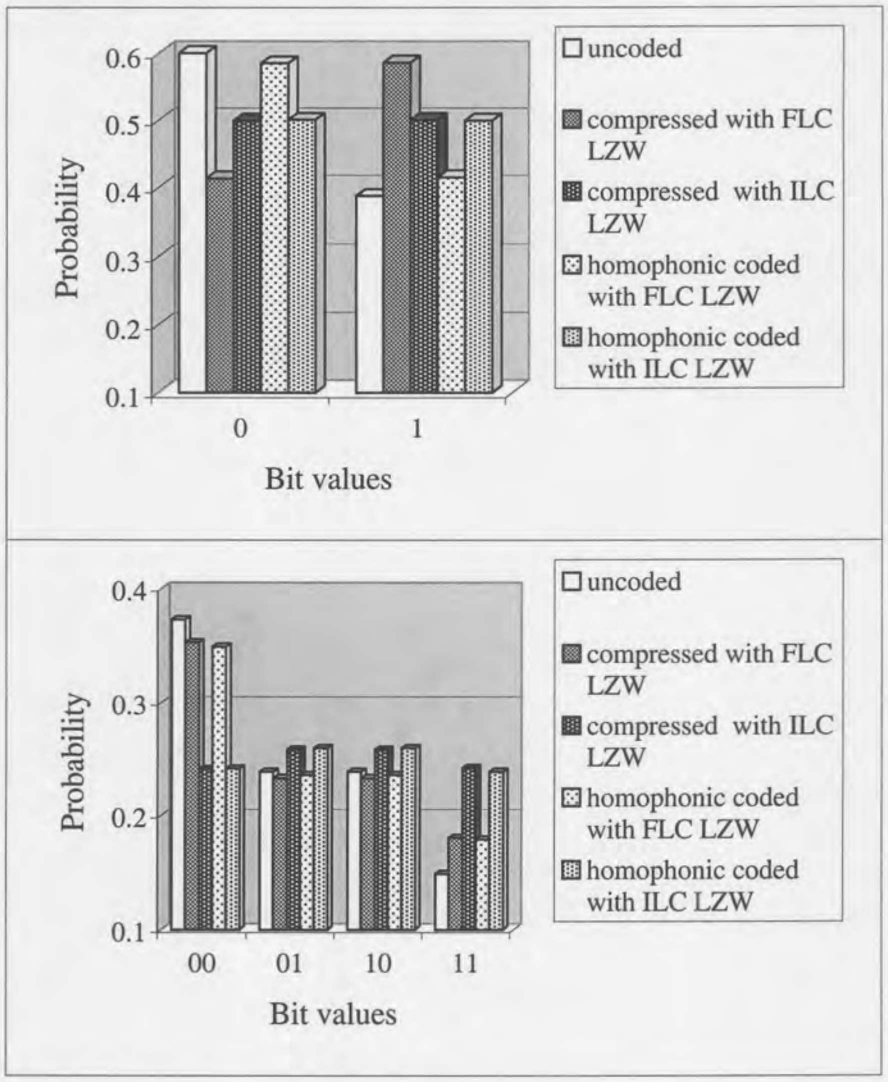


Figure 6.6: Statistics of the uncoded and encoded C++ File

A number of observations can be made from these results. First note that neither the uncoded data, nor the FCL compressed data have a flat frequency distribution of single bits or bit pairs, indicating that the entropy of these two sequences are not a maximum, as is desired. Secondly, note that the occurrence of bits and bit pairs of the FLC homophonic coded data is also not evenly distributed. Even though the input is randomized, denying statistical attacks, the output entropy is not maximized, resulting in a smaller unicity distance of the cipher. The data encoded with ICL homophonic LZW does satisfy the desired requirements, i.e. the homophonic coding algorithms converted the sources into streams in which the number of

1's and 0's are almost equal (equiprobable).

6.3.3 Entropy Results

The goal of the next experiment is to indicate the increase in entropy after files have been encoded. The correct way of comparing the entropy of the input sequence to the entropy of the output sequence would be to consider the input sequence as a collection of symbols $x_1, x_2, \ldots x_n$ of the alphabet X of size n and the output as a collection of symbols $v_1, v_2, \ldots v_m$ of the alphabet V of size m and then use Equation 3.1 to calculate the respective entropies. But since the adapted LZW algorithm does not map specific input symbols to output symbols like traditional homophonic coding, other means have to be resorted to in order to indicate the actual increase in entropy. In the following figure the entropy of the original file and the entropy of the homophonic encoded file (both considered as binary sequences) are plotted against the number of bits encoded. The entropy at a specific point is calculated using the formula for entropy for the binary case:

$$H = p \log_2 \frac{1}{p} + (1-p) \log_2 \frac{1}{(1-p)}, \tag{6.1}$$

where p is the frequency of occurrence of either a 1 or a 0 in a sequence divided by the bit length of the sequence.

To illustrate the effect of local statistics, the "windowing" method described in Appendix H is used to calculate the values of p at a specific point in a file. Fig. 6.7 show the increase in entropy as a result of the homophonic coding, as well as the length of the file before and after it is encoded. This figure show that the homophonic coded sequence have nearly maximum entropy for almost the entire length of the sequence. (Recall that the maximum entropy of a binary sequence is 1 bit/symbol, or 1 bit/bit). It may appear that not much is gained by a small increase in entropy. To see that this is not the case, consider a cipher system that makes use of 56 bit keys. If the entropy of the key is maximum, the unicity distance, given by Equation 3.10, will be 5600 bits if the entropy of the uncoded sequence is 0.99 bits/bit and 56000 bits if the entropy of the encoded sequence is 0.999 bits/bit. A 0.009

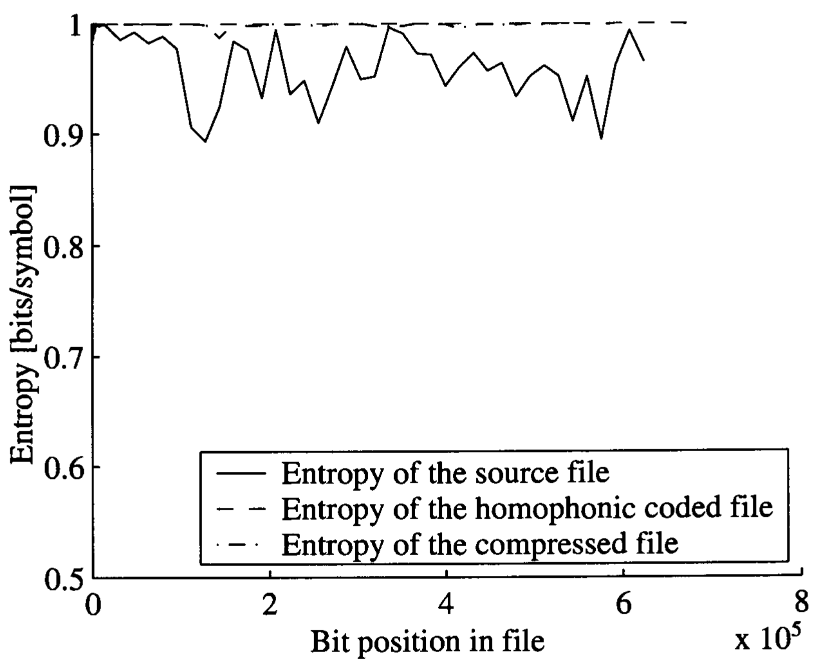


Figure 6.7: Binary entropy of the uncoded, compressed and homophonic encoded C++ file

increase in the source entropy thus resulted in a ten-fold increase in the cipher's unicity distance.

6.3.4 Bits per Symbol Results

Recall that the entropy of a sequence is the minimum number of bits per symbol that are needed to represent the file without loss of information. The goal of a compression algorithm is to represent a file with the least number of bits per symbol. This idea can be used to illustrate the increase in entropy as a result of homophonic coding in another manner. Fig. 6.8 shows the entropy of the C++ file, considered as 8 bit characters, as calculated by the "windowing method". Also shown in the figure, is the entropy of the homophonic encoded file (using the adapted arithmetic coding algorithm) calculated with Eq. 3.1. The number of bits per character used to encode the file with normal arithmetic coding and homophonic arithmetic coding at specific points in the file is also included in the figure. These two graphs show that the entropy of the respective sequences can be approximated with the amount of bits per character used to encode the files up to specific points. These values are calculated by dividing the length of the output file, in bits, by the amount of characters processed from the input file at a particular moment during encoding. The entropy of the normal binary LZW encoded and LZW homophonic encoded files can thus be illustrated by using these approximation methods, also shown in Fig. 6.8.

This figure indicates that the binary LZW algorithm does not compress the file as close to its entropy as one would have expected, especially for small file sizes. The reason for the weak compression at the beginning of the files is that the codebook entries do not comprise long sequences in the beginning of compression, so that the length of the codewords are longer than the source string they represent. It can however be seen that the

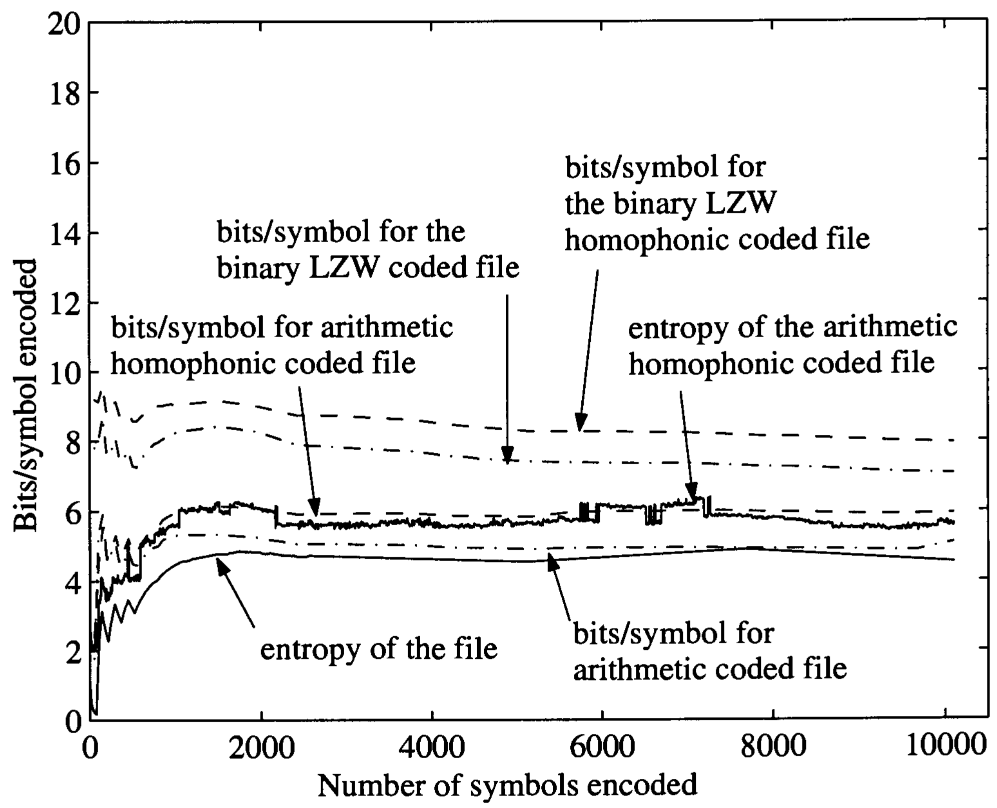


Figure 6.8: Bits per symbol plotted against number of symbols encoded for the compressed and homophonic encoded C++ file

number of bits/symbol at a particular time in encoding is greater for a sequence encoded with the homophonic algorithm than for the same sequence compressed with the LZW algorithm, indicating an increase in entropy.

6.4 Discussion

The results obtained from the experiments performed indicate that the 0/1 balance of encoded files are closer to a 1:1 relation than those of uncoded files. Since the entropy of a sequence is higher when the probabilities of characters appearing are more equally distributed, it indicates that the source entropy is increased after encoding. This results in larger unicity distances which makes a cipher algorithm that encrypt the homophonically encoded sequences stronger than algorithms that encrypt the uncoded sequences. The results also show that the entropy is not increased by more than 2 bits/symbol, in accordance with Eq. 3.16. Furthermore, when comparing the results of the *variable-to-fixed* LZW homophonic coding algorithm to those of the *fixed-to-variable* arithmetic homophonic coding algorithm, it can be seen that the 0/1 balances are more or less equal, but the resulting lengths of LZW encoded sequences are longer than arithmetic encoded sequences. A reason for this might be that the arithmetic coding algorithm estimates the source statistics more accurately than the dictionary based LZW algorithm, or it adapts faster to changing statistics than the LZW algorithm. The entropy figures also illustrate that arithmetic coding approaches the entropy of the source sequences closer than LZW compression. This means that if a source has an entropy of 6 bits/symbol or less, compression is guaranteed for arithmetic coding (if the sequence is long enough), but not for LZW coding.

The results comparing the lengths of encoded files for different modelling methods (Table 6.2) show that when the file is encoded with recency consideration, better compression is always achieved than when it is encoded without recency. This indicates that the source statistics are estimated more accurately, by placing emphasis on more recent characters the algorithm employs a sort of "look ahead" mechanism: the characters following the end of the window are more likely to have statistical properties similar to characters at the end of the window, than those at the beginning of the window.

Chapter SEVEN
Conclusion

The goal of this research was to design an algorithm that is able to convert an arbitrary message source into a uniformly distributed message source without increasing message length. When such sources are encrypted, its unicity distance is higher than when less random sources are encrypted. This means that the number of ciphertext needed to break the encryption algorithm is more for homophonic encoded sources, resulting in stronger security. When all redundancy is removed from a source before it is encrypted, it leads to a cipher system that is *strongly ideal,* because the unicity distance is increased to infinity.

Although data compression algorithms reduce redundancy in sources similar to homophonic coding, homophonic coding is preferred, since it offers the additional advantage that it prevents *known-* and *chosen plaintext attacks* on the encryption algorithms, because the source symbols are randomly mapped onto homophones. This implies that even if a cryptanalyst knows the statistics of the source to be encrypted, this knowledge cannot be used for statistical attacks because the actual homophones that were used are not known. The inclusion of the additional randomness in the sequences when performing homophonic substitution often results in data expansion, hence the search for a method to combine homophonic coding with data compression.

In this dissertation two methods for combining homophonic coding with data compression algorithms were investigated. The two compression algorithms considered were arithmetic coding and LZW.

The arithmetic coding algorithm used a newly introduced probability estimation method based on an IIR filtering scheme. The estimated probabilities were diadically decomposed to form homophones for the input characters, which was encoded with a faster arithmetic coding algorithm called *shift-and-add* arithmetic coding.

The LZW algorithm was adapted so that the binary source is randomized during compression by making use of a random source and exclusive-or operations.

Both homophonic algorithms were able to randomize the different sources used in the experiments, as indicated by the results. The arithmetic coding algorithm achieved better overall compression and guarantees compression if the source has an entropy of not more than approximately 6 bits/symbol (and is sufficiently long).

Because both compression algorithms are adaptive, compression is not guaranteed if the sequences are short - the algorithms must first adapt to the source statistics before it is accurately modeled and compression can occur. The arithmetic coding algorithm always adapts faster than the LZW algorithm, which means that it requires less characters than LZW to guarantee compression.

7.1 Proposals for Further Research

Further research can be conducted in the fields of:

- The order of statistical modeling for arithmetic coding. A 0-order model was used in this study, where higher order models may achieve better compression.
- Langdon and Rissanen [17] illustrate a binary version of arithmetic coding, which considers the source to be binary and divides the interval [0,1) into only two intervals.

Experiments can be performed to investigate the effect of the dyadic decomposition method of homophonic coding using such an algorithm. This could result in a high speed encoding algorithm, because bit probability estimation can be performed much faster than character probability estimation, and if the two intervals are dyadically decomposed, the fast shift-and-add method of arithmetic coding can be implemented.

REFERENCES

[1] "Building an E-Commerce Trust Infrastructure: SSL Server Certificates and Online Payment Services," *technical brief, available at http://www.verisign.com.*

[2] W. Stallings, *Cryptography and Network Security.* Prentice Hall, 2 ed., 1999.

[3] C. E. Shannon, "Communication Theory of Secrecy Systems," *Bell Syst. Tech. J.*, vol. 28, pp. 656–715, 1949.

[4] J. L. Massey, "Some Applications of Source Coding in Cryptography," *European Transactions on Telecommunications*, vol. 5, pp. 7/421–15/429, July-August 1994.

[5] C. G. Günter and A. B. Boveri, "A Uninversal Algorithm for Homophonic Coding," *Advances in Cryptology-Eurocrypt 1988, Springer-Verlag*, pp. 405–414, 1988.

[6] I. H. Witten and J. G. Cleary, "On the Privacy Afforded by Adaptive Text Compression," *Computers and Security*, vol. 7, pp. 397–408, 1988.

[7] C. Smith, "Adaptive Homophonic Coding of Cryptographic Sources," Master's thesis, University of Pretoria, October 1998.

[8] M. Nelson and J. L. Gaily, *The Data Compression Book.* M&T Books, 1995.

[9] W. Stallings, *Internet Security Handbook.* McGraw-Hill, 1995.

[10] J. Seberry and J. Pieprzyk, *Cryptography An Intorduction to Computer Security.* Prentice Hall, 1988.

[11] C. Brenton, *Mastering Network Security.* Network Press, 1996.

[12] W. F. ans M S Baum, *Secure Electronic Commerce.* Prentice Hall, 1997.

[13] "SSL Specification," *latest copy available at http://www.netscape. com.*

[14] J. G. Proakis, *Digital Communications.* McGraw-Hill International, 4 ed., 2001.

[15] H. N. Jendal, Y. J. B. Kuhn, and J. L. Massey, "An information-Theoretic Treatment of Homophonic Substitution," *Advances in Cryptology - Eurocrypt '89 Lecture Notes in Computer Science, Springer*, pp. 382–394, 1990.

[16] C. Boyd, "Enhancing Secrecy by Data Compression: Theoretical and Practical Aspects," *Advances in Cryptology-Eurocrypt 1991*, no. 547, Springer Verlag, pp. 266–280, 1991.

[17] G. Langdon, "An Introduction to Arithmetic Coding," *IBM J. Res. Develop.*, vol. 28, no 2, pp. 135–149, March 1984.

[18] G. Langdon and J. Rissanen, "Compression of Black-White Images with Arithmetic Coding," *IEEE trans. Commun.*, vol. COM-29, pp. 858–867, June 1981.

[19] G. Langdon and J. Rissanen, "Arithmetic Coding," *IBM J. Res. Develop.*, vol. 23, pp. 149–162, March 1979.

[20] T. C. Bell, J. G. Cleary, and I. H. Witten, *Text Compression.* Prentice Hall, 1990.

[21] P. G. Howard and J. S. Vitter, "Analysis of Arithmetic Coding for Data Compression," *Information Processing and Management*, vol. 28, no. 6, pp. 749–764, 1992.

[22] P. Elias, "Interval and Recency Rank Coding: Two On-line Adaptive Variable Length Schemes," *IEEE Trans. Inform. Theory IT-33*, pp. 3–10, Jan 1987.

[23] W. T. Penzhorn, "A Fast Homophonic Coding Algorithm Based on Arithmetic Coding," *Fast Software Encryption: Second International Workshop*, vol. 1008, pp. 329–345, 1994.

[24] T. Welch, "A Technique for High-Performance Data Compression," *IEEE Computer*, vol. 17, Number 6, pp. 8–19, June 1984.

[25] W. T. Penzhorn, “Homophonic Substitution Cipher,” *Private communications*, 1993.

[26] M. R. Nelson, “Lzw Data Compression,” *Dr. Dobb's Journal*, pp. 29–36, October 1989.

APPENDIX A
IMPLEMENTATION

A.1 Introduction

In order to show that the two described homophonic coding algorithms do indeed convert a practical source with a non-uniformly distributed alphabets into a "random" binary source, computer simulations are needed. The C++ programming language was used to write programs that implement the described encoding algorithms utilizing real-life data. The reason for choosing C++ as the programming language is due to its efficiency and for allowing programmers more control over the hardware, so that low level operations e.g. bit shifting can be performed at high speeds.

A.2 Homophonic Coding with Arithmetic Coding

A shift-and-add homophonic arithmetic coding algorithm was implemented with 0-order *adaptive* modeling. The four steps of the arithmetic encoding procedure, as described in Section 4.4, are performed as follows:

A.2.1 Step1: Source Modeling

The adaptive modeling procedure utilizes the frequency of occurrence of past characters to estimate the probabilities of occurrence of future characters that are to be processed. At the start of the encoding, no characters have been processed and the model is assigned an initial state. The source input is considered to be 8-bit ASCII characters. This means that there are $2^8 = 256$ possible symbols in the source alphabet. As mentioned in Section 4.3.3, an end-of-file symbol also has to be included in the model. In total, there are thus 257 symbols in the model. During initialization, the model assigns equal probabilities to all symbols, i.e. each arithmetic coding interval width is proportional to 1/257. This is done by means of the q components in $\hat{P}_i$. (All initial filter outputs are equal to 0, and according to Eq. 4.4, this results in $\hat{P}_i(0) = (0+q)/(0+Lq) = 1/L = 1/257 \quad i = 1, 2, \ldots, 257$).

The q value also prevents an underflow, because even if the value of $Y_i(k)/(T_k + Lq)$ becomes to small to fit in the register, the non-zero q component is always added in the estimation of $\hat{P}_i(k)$. The program uses 16-bit unsigned integers to store the estimated probabilities. In this case, q can be calculated using Eq. 4.9

$$q = \frac{1}{(1-\alpha)(2^B - L)} = \frac{1}{(1-\alpha)(2^{16} - 257)}$$

When computing T_k, it is not necessary to calculate the sum of all the filter outputs at each time instance. According to Eq. 4.7, T_k can be calculated by

$$T_k = \alpha T_{k-1} + 1, \tag{A.1}$$

where $T_0 = 0$. In the C++ implementation, an array called `charwidths` is used to store the 257 values of Y_i, and a variable called `Tk` is used to store the current value of T_k. After each symbol encoded, every value in this array, as well as `Tk`, is multiplied with α. The value in the position in the `charwidths` array corresponding to the current symbol being encoded, as well as the value of `Tk`, is then incremented with 1. The estimated probabilities are then given by Eq. 4.4:

$$\hat{P}_i(k) = \frac{Y_i(k) + q}{T_k + Lq} = \frac{\texttt{charwidths[si]} + \frac{1}{(1-\alpha)(2^{16}-257)}}{\alpha\texttt{Tk[k - 1]} + \frac{257}{(1-\alpha)(2^{16}-257)}} \qquad i = 1, 2, \ldots, L$$

A.2.2 Step2: Design of the Homophonic Channel

Instead of performing the division process in Eq. 4.4 when a symbol's probability is calculated, and converting the result to an integer, and dyadically decomposing the integer to calculate the homophones assigned to each symbol, these three steps are combined for high speed encoding performance. The way to achieve this is as follow:

Note that the binary result of a/b, where $a < b$, can be obtained by repeating the following two steps until $a = 0$, or the required precision is reached:

Step 1:

Multiply a by 2. (left shift a with 1 bit).

Step 2:

If $a \geq b$ output 1 and subtract b from a.

If $a < b$ output 0.

For example, to represent 5/8, the steps would be:

$5 \times 2 = 10 > 8 \ \therefore$ output 1: 0,1

$a = 10 - 8 = 2$

$2 \times 2 = 4 < 8 \ \therefore$ output 0: 0,10

$4 \times 2 = 8 \ \therefore$ output 1: 0,101

$a = 8 - 8 = 0 \ \therefore$ terminate.

The result of 5/8 is thus 0,101.

The cumulative frequency counts of the characters are stored in a variable named `CumFreq`, which is a 16-bit unsigned integer (initialized with 0). 5/8 will thus be stored as 1010000000000000. This implies that the steps should be repeated until $x = 0$, or the bit-length of the result is equal to 16. The homophones for a character are obtained by taking the entries of the `charwidths` array, adding the value of q to it, and dividing the result by (`tk`+Lq) using the two steps described above. If the output is a 1 after the i-th round, the particular codeword for that homophone is equal to `CumFreq` and the value of 2^{16-i} is added to `CumFreq`. The probability associated with that particular homophone is equal to 2^{-i}. The number of homophones associated with each symbol is equal to the number of 1's in the 16 bit result of the "division" process. Refer to Appendix E for an example of this procedure.

A.2.3 Step 3: Random Selection of Homophones

When randomly choosing the specific homophone to represent the particular source symbol, homophones with shorter lengths (higher probabilities) must be chosen more frequently than those with longer lengths (lower probabilities). The *a posteriori* probabilities of the homophones must thus also be taken into account. For example, if the probability of a particular source symbol is P(s_i)=5/8, its two homophones will have probabilities of P(v_{i1})=4/8 and P(v_{i2})=1/8 respectively. The first homophone must thus be chosen P($v_{i1}|s_i$)=(4/8)/(5/8)=4/5 of the time and the second P($v_{i2}|s_i$)=(1/8)/(5/8)=1/5 of the time. This is achieved by utilizing a random number generator that generates numbers that are uniformly distributed between 0 and a maximum value `max`. The interval (0-`max`) is divided into subintervals with widths proportional to the *a posteriori* probabilities of the codewords. When the generated random number falls in a specific subinterval, the homophone associated with that subinterval is encoded.

A.2.4 Step 4: Arithmetic Coding of the Homophones

To store the adaptive model, the program uses a structure array called `Symbols` with three members: `char` to store the original character that is to be associated with the homophones, `CumFreq` to store the cumulative probabilities of the homophones and `Length` to store the number of bits that the next homophone to be encoded have to be shifted with in the shift-and-add procedure. The value of `Length` is equal to i, where i is equal to the round number in the "division" process. Refer to Appendix E for an example of this procedure.

In order to prevent an overflow, the result of the shift-and-add procedure is stored in a 64-bit buffer. As soon as the bit-length of the shift-and-add result exceeds 48 bits, the first 16 bits are output. Appendix G illustrates part of a log file that was kept during an encoding simulation. The log file comprises the 64-bit buffer values calculated during encoding. As can be seen from its output, by the time the buffer length exceeds 48, the 16 most significant bits do not undergo any more changes. Bit stuffing is also used to prevent overflow by inserting a 0 as soon as 16 consecutive 1's have occurred. The adaptive homophonic arithmetic encoding algorithm is shown in Fig. A.1.

The initialization procedure initiates the model by setting all the entries in the `Charwidths` array equal to 0, and calling the `CalculateHomophones` subroutine. It also calculates the value of q. The `CalculateHomophones` subroutine multiplies each entry in the `Charwidths` array with α and calculates the new value of `Tk` from the current `Tk`. It then performs the "division" process to calculate the codewords associated with each input symbol (the cumulative probabilities of the homophones) and the shifting length `Length` associated with each codeword (where `Length` $= -\log_2 P(\text{homophone})$). `SelectHomophone` is a subroutine that randomly selects a codeword to represent a source symbol in the manner described in the previous section, and returns the index position of the codeword in the `Symbols` array to a variable called `hom`.

The initial value of the 64 bit buffer is set to 0. After the last symbol has been read in, the encoder performs one last shift-and-add operation, i.e. that of the codeword for the `EOF` character. Before this is done, care must be taken that the bit length of the buffer is less than 48, to prevent an overflow. The final value of the buffer is then broken up into parts of 16 bits long, and is output.

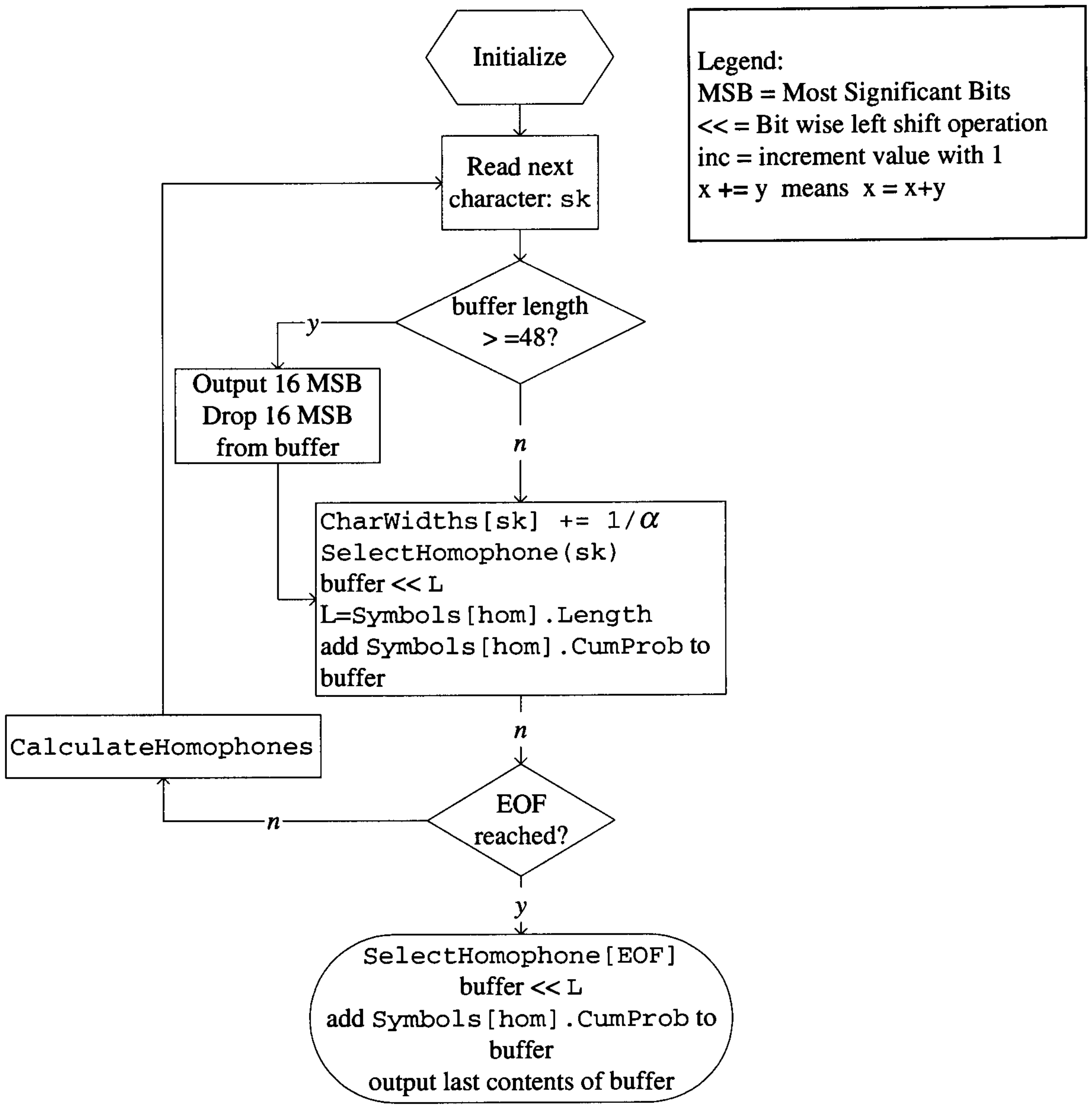

Initialize
Read next character: sk
buffer length > =48?
y
n
Output 16 MSB Drop 16 MSB from buffer
CharWidths[sk] += 1/α
SelectHomophone(sk)
buffer << L
L=Symbols[hom].Length
add Symbols[hom].CumProb to buffer
CalculateHomophones
n
EOF reached?
n
y
SelectHomophone[EOF]
buffer << L
add Symbols[hom].CumProb to buffer
output last contents of buffer
Legend:
MSB = Most Significant Bits
<< = Bit wise left shift operation
inc = increment value with 1
x += y means x = x+y

Figure A.1: Program flow chart for the adaptive homophonic arithmetic encoding algorithm

A.2.5 Decoding

The decoding of a sequence encoded with the above described algorithm amounts to comparison and subtraction operations that are repeated until the `EOF` character is received.

A.2.5.1 Comparison Procedure

The decoder starts with the same model as the encoder, and updates the model as the characters are being received. The same `Symbols` structure array is used as was used in encoding. At any point during decoding, this array is the same as it was at the same point during encoding. After the first 16 bits are received, they are stored in a buffer and the value is compared to the cumulative probabilities in the `Symbols` structure array. The received character is the one corresponding to the interval of the homophone in which the received value lies.

A.2.5.2 Subtraction Procedure

The cumulative probability of the identified homophone is subtracted from the received value, and the result is shifted left with the value of `Length` associated with that homophone. The same number of bits from the next received 16 bits are shifted into the buffer. After the character has been identified, the model is updated. The decoding process terminates the moment when the received character is the `EOF` character.

A.3 Homophonic Coding with LZW

The LZW implementation described by Nelson *et al.* [8] was used as the basis for the LZW homophonic coding algorithm in this study. Nelson's 12 bit fixed dictionary implementation was adapted by changing the codebook size to 14 bits and using a maximum table size of 20483, the first prime number larger than 1.25×2^{14}. Instead of using 14 bits to represent each codeword, the codeword size was incrementally increased, according to the method described in Section 5.3.

A.3.1 Encoding

The implementation of Nelson assumes that the input symbol sequence is ASCII characters, where, for the purposes of this study, it would be better to consider it as a binary string. (Recall from Chapter 3 that the entropy of a source can be approached better when the source consists of low alphabet sizes). In order to convert Nelson's algorithm to a binary LZW algorithm, each character was decomposed into bits by performing a bit-wise *and* operation with the numbers $2^7, 2^6, 2^5, \ldots, 2^0$ and each character. The results are taken as the input to the binary LZW algorithm. For example, if the first character is an "A", corresponding to the ASCII value of 65, or 01000001 in binary, the first input bit becomes:

$$\begin{array}{r} 01000001 \\ \&\ 10000000 \\ \hline 0 \end{array}$$

where & is the bit-wise *and* operation. The second input bit becomes:

$$\begin{array}{r} 01000001 \\ \&\ 01000000 \\ \hline 1 \end{array}$$

and so forth. In order to accomplish this, a loop is added to their main loop that decomposes each input character into bits in this manner.

Another aspect of Nelson's implementation that needed to be changed is that instead of reading the input on a character-by-character basis from the input file, the entire file must be read into an array beforehand. This is needed in order to perform the randomization procedure on the input. A subroutine called `RandomizeRest` is inserted in the code just before the output codeword is to be written to disk, to randomly choose a bit and perform the randomization procedure on the remainder of the input sequence. This subroutine returns the chosen random bit so that it can be added to the output stream. Fig. A.2 describes the entire encoding algorithm. The program uses an array structure called `dict`, to store the dictionary/codebook entries. This structure comprises of three members, namely `code` to

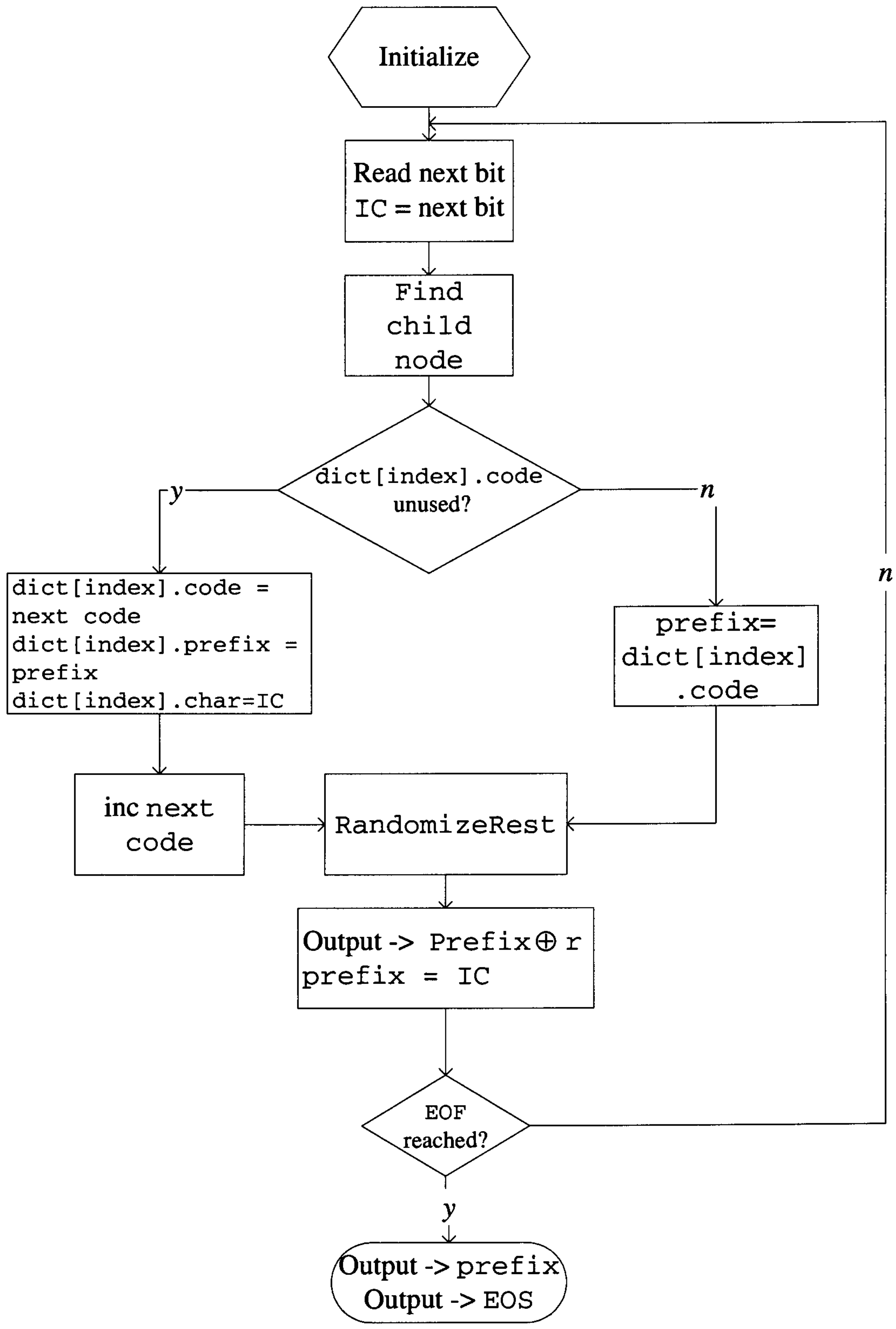


Figure A.2: Program flow chart for the adapted LZW encoding algorithm

store the codebook entry number, `prefix` to store the prefix of that particular entry and `char` to store the Innovation Character. The size of this array is 20483. In order to conserve space in Fig. A.2, it is assumed that the input characters are already converted into its binary form.

The initialization procedure takes care of a number of things. First it sets all the codebook `code` values equal to -1, which represents them as *unused*. It initializes `next code` to 3, the first phrase to be output, and sets the End of Stream value (`EOS`) equal to 2. Then it reads the first input bit and set `prefix` equal to it.

The subroutine `Find Child Node` takes two parameters as input, namely the `prefix` and `IC`. It performs the hashing function described in Section 5.2.3, and returns a value called `index` to the main program. This is the index in the structure array where that codebook entry is either unused, or contains the same value of `prefix` and `IC` as those input into the subroutine.

A.3.2 Decoding

The decoding program makes use of a subroutine called `decode string` to decode a received codeword and an array called `decode stack` to store the decoded characters in. The `decode string` subroutine takes two parameters: the codeword to be decoded and a variable called `count` that counts the number of characters decoded for the specific codeword. This value is also returned by the subroutine to indicate how much characters should be read from the `decode stack` array. The decoding program also builds up the (exact same) codebook as it decodes in the structure array `dict`.

When a codeword is received, the `decode string` subroutine stores the innovation character of the codeword in the first position of the `decode stack` array, and then goes to the position in the dictionary that is equal to the prefix of the codeword. The innovation character of this entry is then stored in the second position of the `decode` `stack` array and the process is repeated until the received codeword has been decoded entirely. The `decode stack` array thus contains the decoded string in reverse order. The implementation of Nelson *et al.* [8] writes the contents of the array to disk on a character-by-character basis in the reverse order after it has been constructed. In the binary implementation used here the contents of the array is stored in a buffer (in reversed order) and is not written to disk until the buffer comprises of 8 bits, whereafter it is written to disk as a character.

The exception handler that takes care of the problem that a not yet fully constructed dictionary entry has to be decoded, is inserted in the beginning of the main loop. This is the reason why the variable `count` is given as input to the `decode string` subroutine. When the received codeword is not yet fully constructed, `count` is set to 1, (where it is normally set to 0) and the first entry of the `decode stack` array is set equal to the current bit character, which was the last entry in the `decode stack` array i.e. the first bit in the current prefix. Instead of sending the received codeword to the `decode string` subroutine, the exception handler sends the previously received codeword, and this completes the exception handling process.

The decoding of the data constructed by the adapted encoding algorithm consists of properly executed *xor* operations to undo the *xor* operations of the encoder. This is achieved in the following manner. After the decoder has received and decoded the first codeword, it obtains the first random bit inserted by the encoder. This random bit is stored in a variable called `randombit`. The decoder then reads and decodes the next received codeword as normal. Then the decoder performs a *xor* operation with the first random bit (stored in `randombit`) and the first bit in the current (third) output sequence. `randombit` is then *xor*red with the second random bit obtained from the stream, and the remainder of the current (third) output sequence is *xor*red with the result on a bit by bit basis. The first bit of the next (fourth) output sequence is also *xor*red with `randombit`. The third random bit is *xor*red with `randombit` and the result is used for the *xor* operation on the remainder

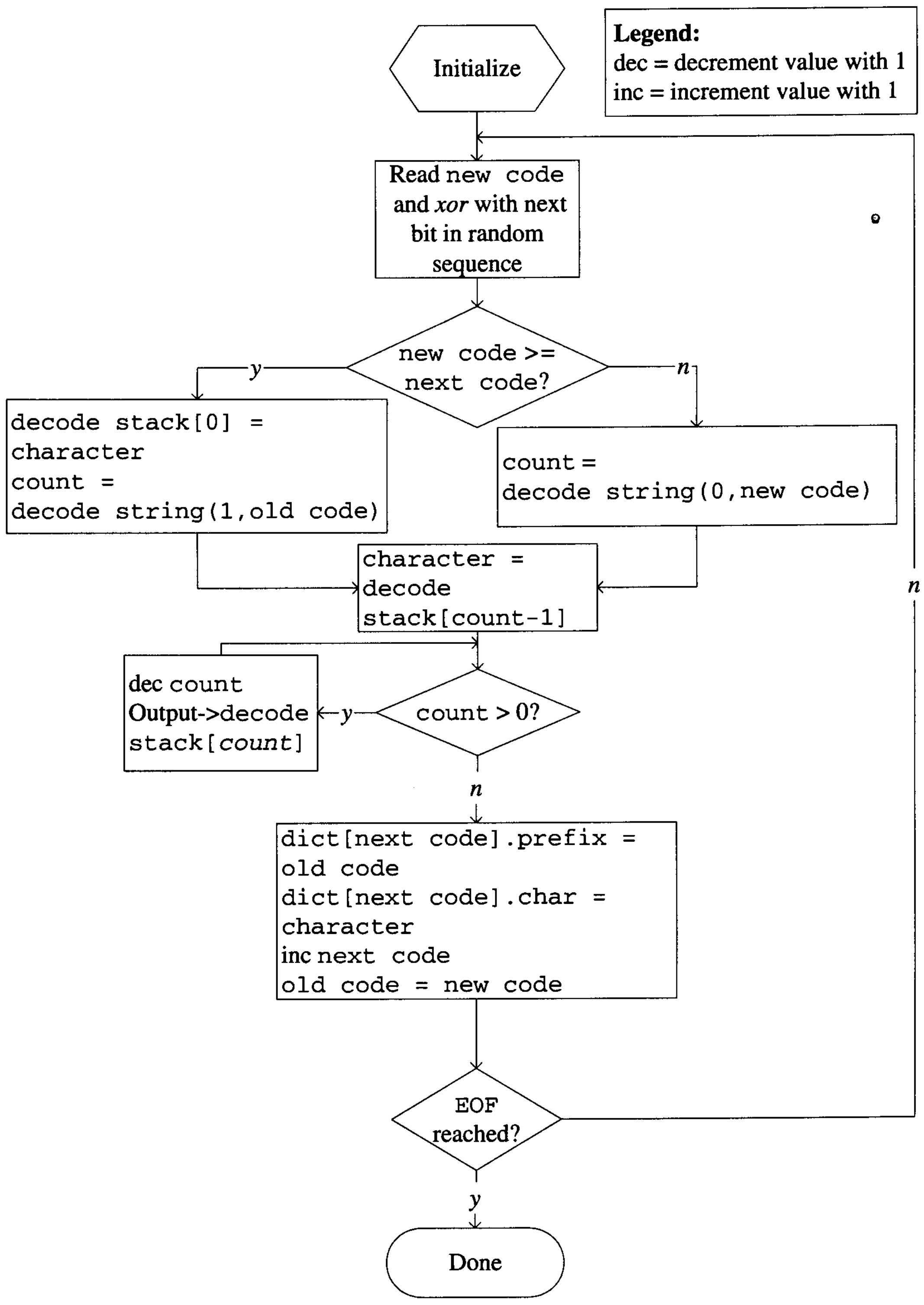

Legend:
dec = decrement value with 1
inc = increment value with 1
Initialize
Read new code and xor with next bit in random sequence
new code >= next code?
y
n
decode stack[0] = character
count = decode string(1,old code)
count = decode string(0,new code)
character = decode stack[count-1]
dec count
Output->decode stack[count]
count > 0?
y
n
dict[next code].prefix = old code
dict[next code].char = character
inc next code
old code = new code
EOF reached?
n
y
Done

Figure A.3: Program flow chart for the adapted LZW decoding algorithm

of the (fourth) output sequence. This procedure is then repeated for the entire decoding process. It is the task of the `decode string` subroutine to perform these *xor* operations. Refer to Appendix F for an example. Fig. A.3 illustrates the flowchart of the decoding

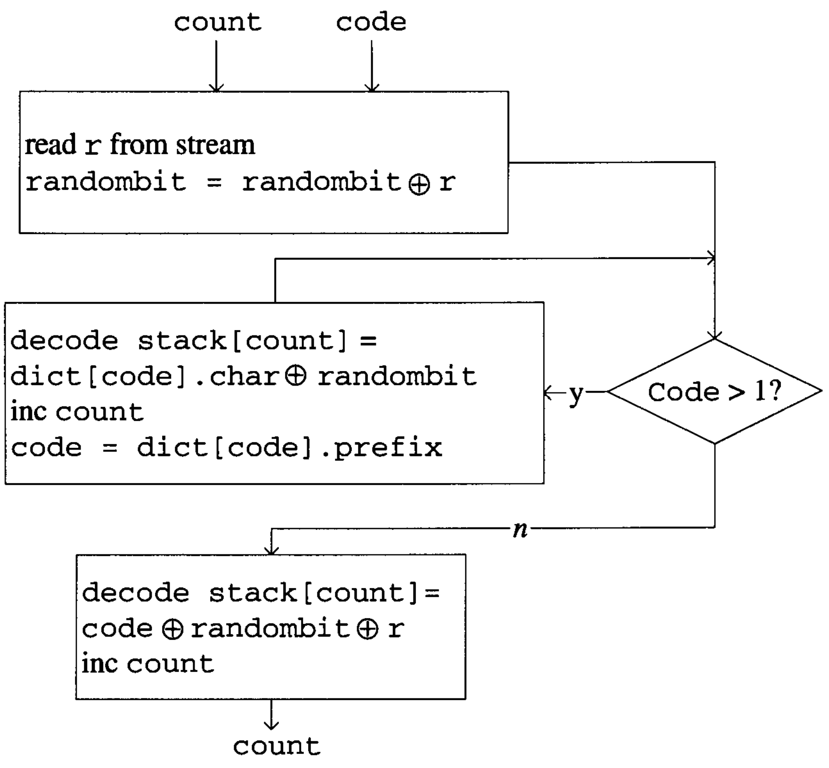


Figure A.4: Program flow chart for the decode string subroutine

process. Once again it is assumed that `Output->` takes care of converting 8 received bits into characters. The initialization procedure sets the variable `next code`, which is used to store the current dictionary entry in, equal to 3, the first code value. It also sets `character` equal to the first received codeword *xor*red with the first bit in the random sequence. The operation of comparing the `new code` with `next code` is the exception handler. Fig. A.4 illustrates a flowchart for the operation of the `decode string` subroutine. `r` is the current random bit obtained from the received bit stream.

APPENDIX B
SOURCE FILES USED IN EXPERIMENTS

The five source files used in the experiments were as follows:

- HTML file: The source of the home page for www.ecommercetimes.com on the 8th of September 2001.
- C++ file: The source code of the adaptive homophonic arithmetic encoder program used to perform the experiments.
- TEX file: The TEX file of this document.
- English literature file: Chapter 1 of the book *The Hobbit* by J.R.R. Tolkien
- Electronic transaction data: This is obtained by purchasing a book from www.Amazon.com. When the page that asks for the credit card number appears, the html source can be saved and the destination of the form data as specified by the source can be altered to send the information to a local machine instead of www.Amazon.com.

APPENDIX C

RESULTS OF HOMOPHONIC CODING WITH ARITHMETIC CODING

This Appendix show the Arithmetic coding results that was obtained for the other files used in the experiments. The same experiments were performed as was done for the C++ file in Chapter 6. Fig. C.1 to Fig. C.4 shows the entropy plots for the various source files.

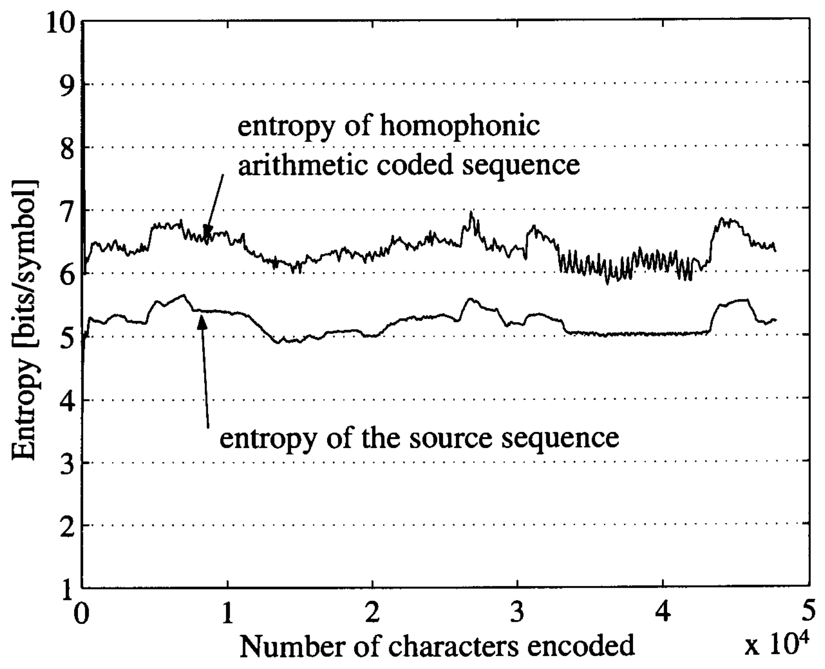


Figure C.1: Entropy of the source and homophonic encoded html file

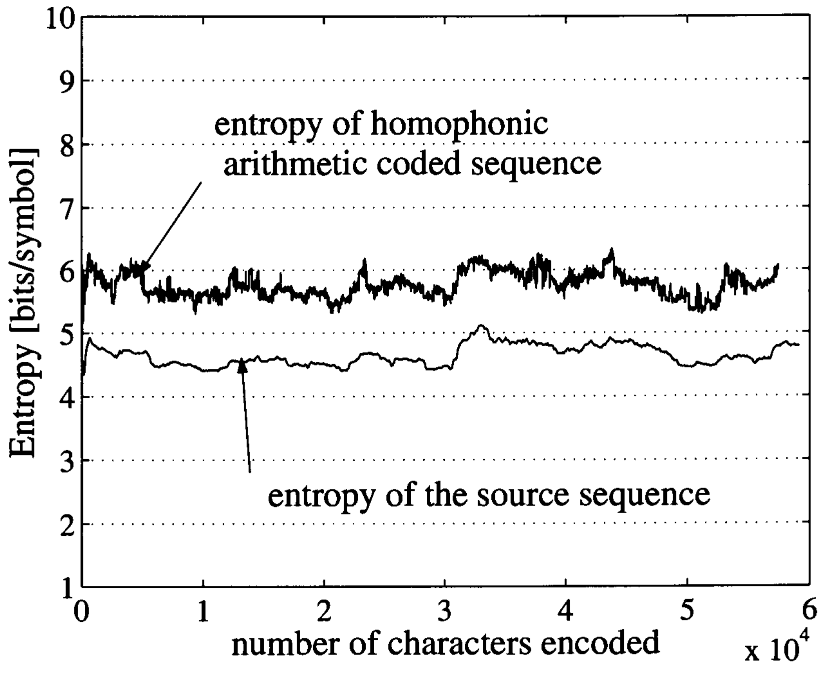


Figure C.2: Entropy of the source and homophonic encoded TEX file

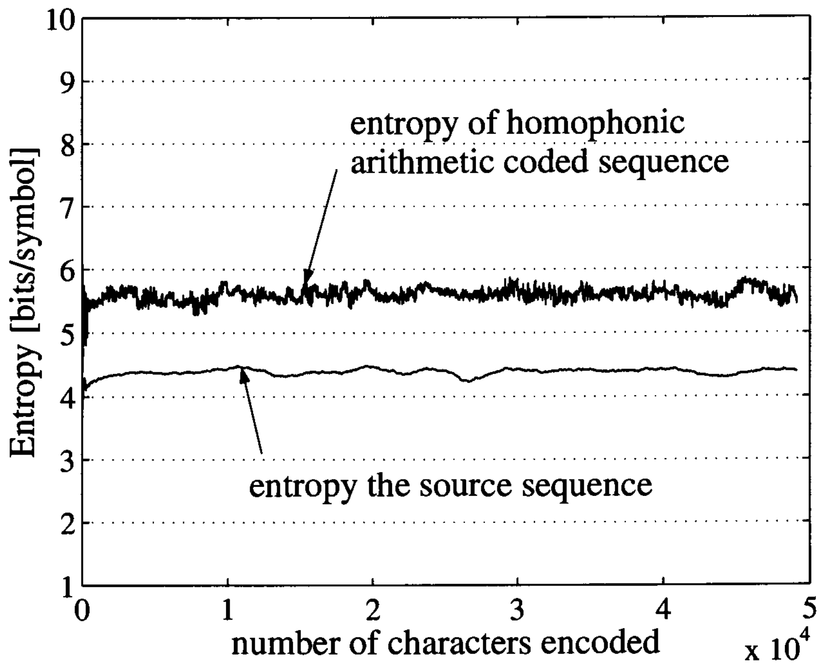


Figure C.3: Entropy of the source and homophonic encoded English text file

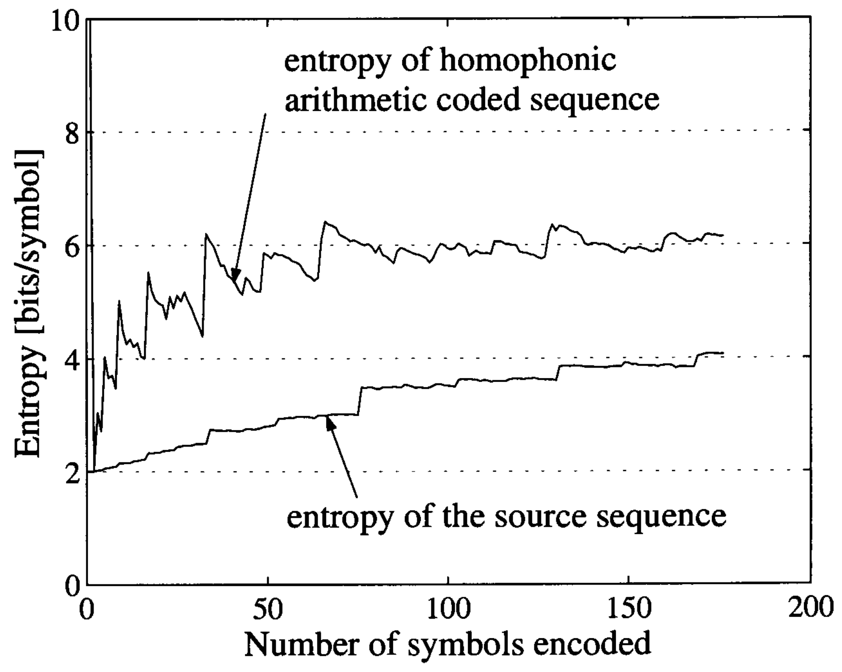


Figure C.4: Entropy of the source and homophonic encoded e-commerce data

Fig. C.5 to Fig. C.8 show the statistics of the uncoded files, and files encoded with the homophonic arithmetic coding algorithm when statistic modelling was done with the IIR based method, and for $\alpha = 0.999$. Table C.1 to Table C.4 show the compression results for the files when encoded with the different statistical modelling techniques.

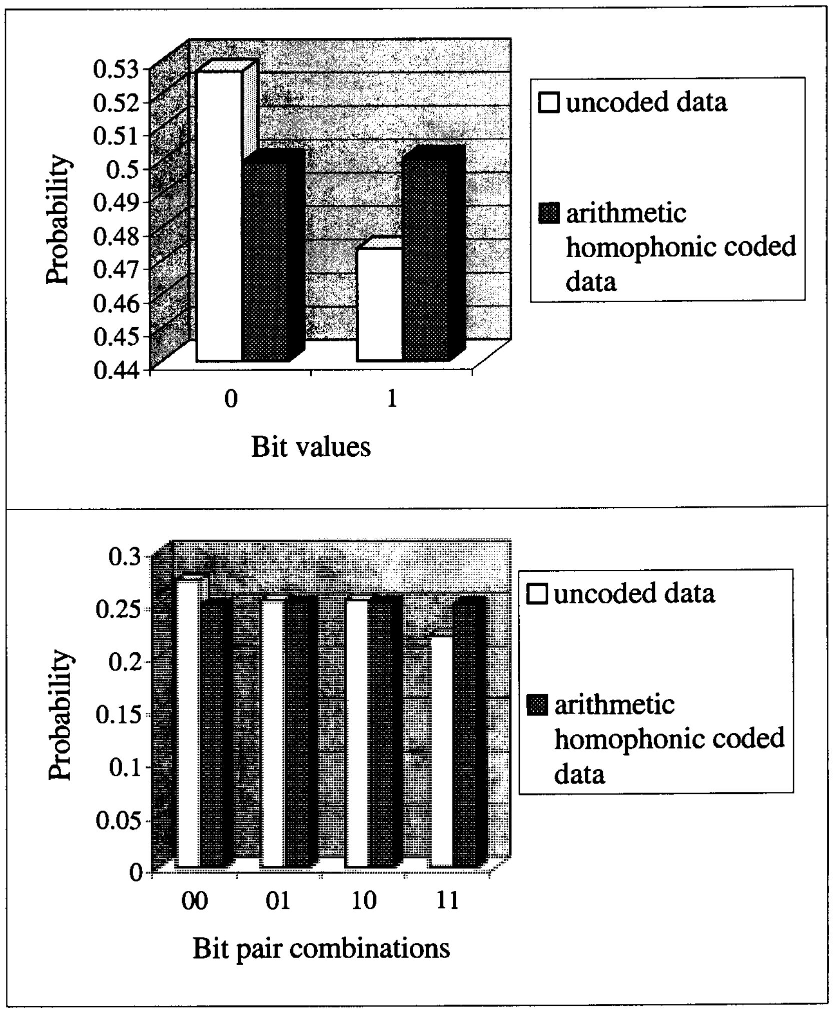


Figure C.5: Statistics of the uncoded and encoded html file

TABLE C.1: **Compression results for the html file for various probability estimation methods**

Method	Length of original file	Length of encoded file	Percentage of original length
SW Method	47705 bytes	38930 bytes	81.61%
VW Method	47705 bytes	38890 bytes	81.52%
LA Method	47705 bytes	38204 bytes	80.08%
IIR Method	47705 bytes	38126 bytes	79.92%

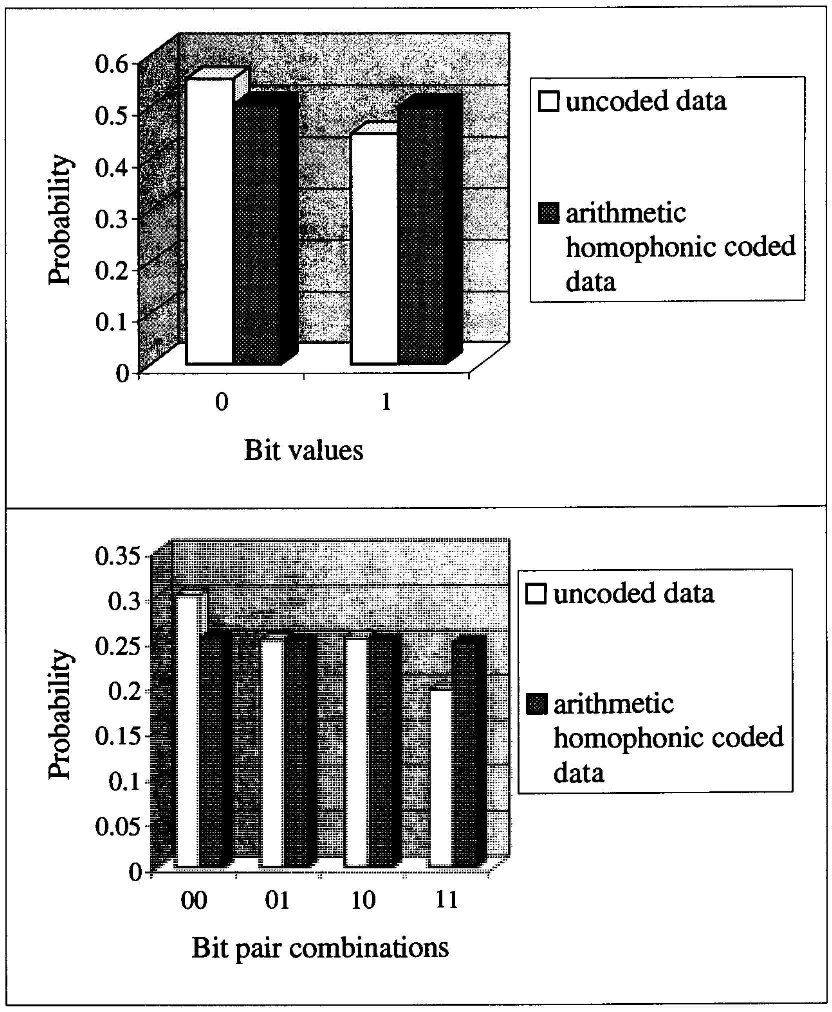


Figure C.6: Statistics of the uncoded and encoded TEX file

TABLE C.2: **Compression results for the TEX file for various probability estimation methods**

Method	Length of original file	Length of encoded file	Percentage of original length
SW Method	59139 bytes	43770 bytes	74.01%
VW Method	59139 bytes	43414 bytes	73.41%
LA Method	59139 bytes	42762 bytes	72.31%
IIR Method	59139 bytes	42666 bytes	72.15%

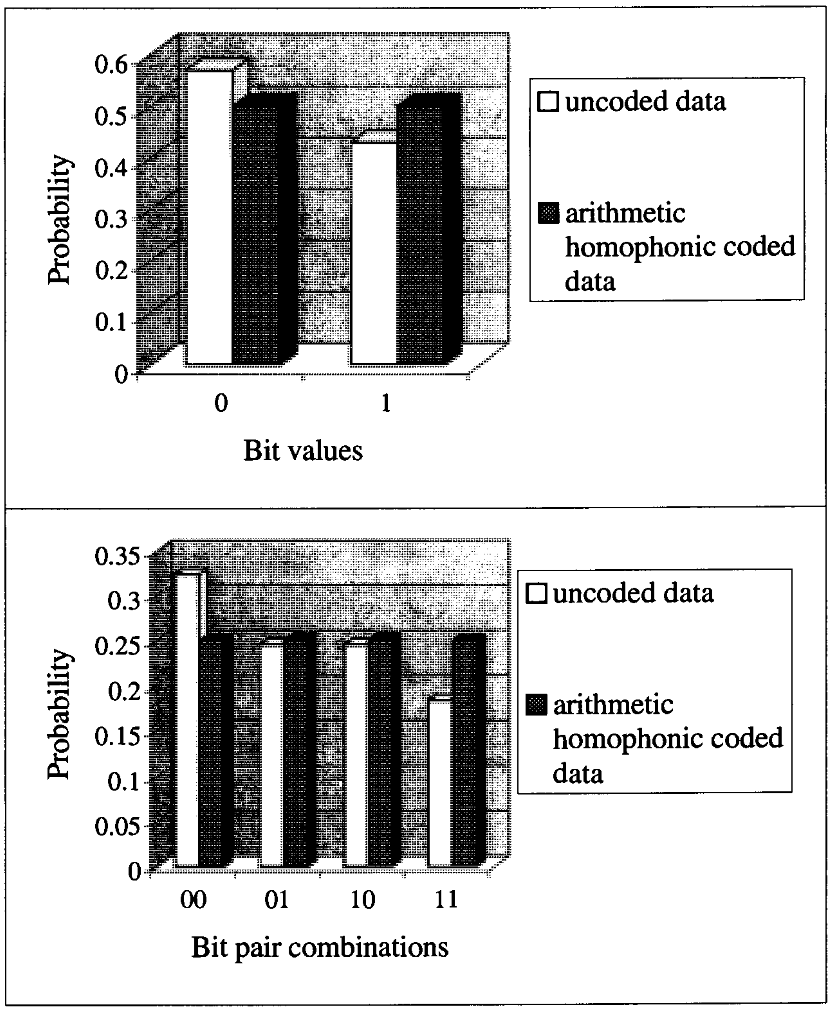


Figure C.7: Statistics of the uncoded and encoded English text file

TABLE C.3: **Compression results for the English literature file for various probability estimation methods**

Method	Length of original file	Length of encoded file	Percentage of original length
SW Method	49098 bytes	34910 bytes	71.10%
VW Method	49098 bytes	34904 bytes	71.09%
LA Method	49098 bytes	34326 bytes	69.91%
IIR Method	49098 bytes	34320 bytes	69.90%

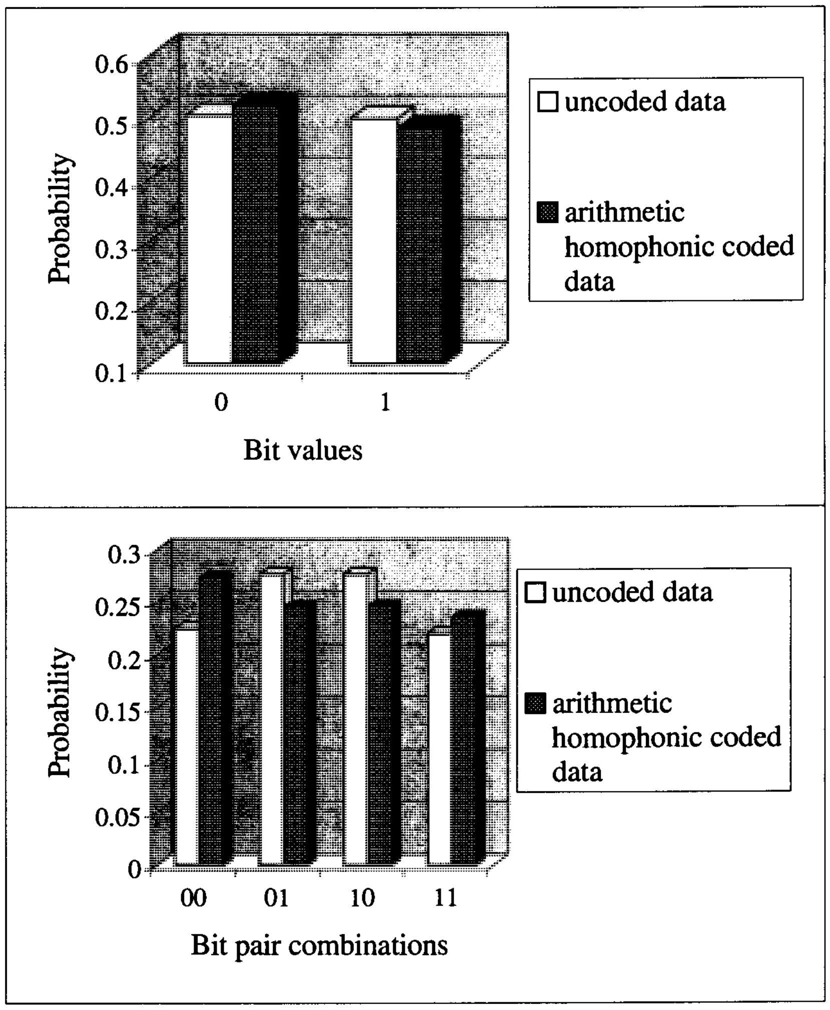


Figure C.8: Statistics of the uncoded and encoded e-commerce data

TABLE C.4: **Compression results for the e-commerce data for various probability estimation methods**

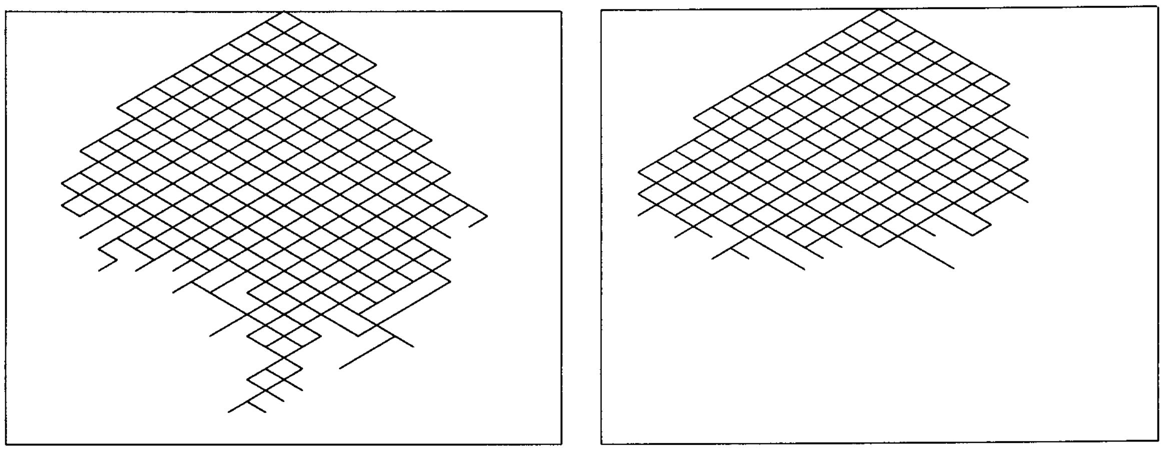
Method	Length of original file	Length of encoded file	Percentage of original length
SW Method	176 bytes	178 bytes	101.14%
VW Method	176 bytes	178 bytes	101.14%
LA Method	176 bytes	180 bytes	102.27%
IIR Method	176 bytes	164 bytes	93.18%

Appendix D

Results of Homophonic Coding with the LZW Algorithm

This Appendix show the LZW results that was obtained for the other files used in the experiments. The same experiments were performed as was done for the C++ file in Chapter 6.

D.1 Bit Patterns



(a) (b)

Figure D.1: Binary LZW trees of dictionary entries formed when encoding (a) the HTML file with normal LZW and (b) HTML file with homophonic LZW

Fig. D.1 (a) shows the binary tree obtained when compressing the `html` file with the binary LZW algorithm. Fig. D.1 (b) shows the binary tree obtained when encoding the same file with the homophonic LZW algorithm. Fig. D.2 (a) and Fig. D.2 (b) show the results obtained for the TeX file. Fig. D.2 (c) and Fig. D.2 (d) show the results obtained for the English literature text file and Fig. D.3 (a) and Fig. D.3 (b) show the results obtained for the e-commerce data. In all these plots the tree on the left hand side show the normal compressed

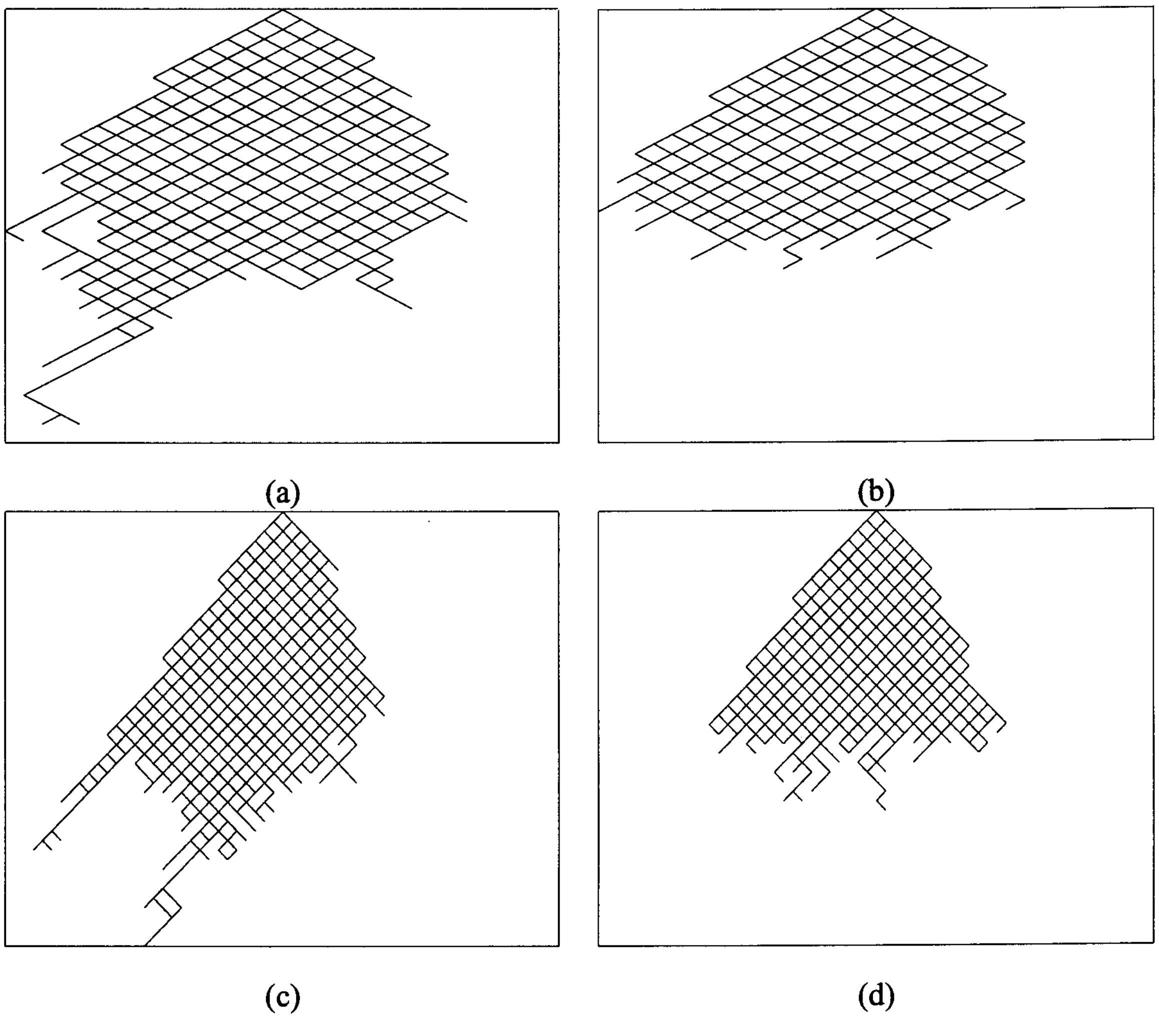
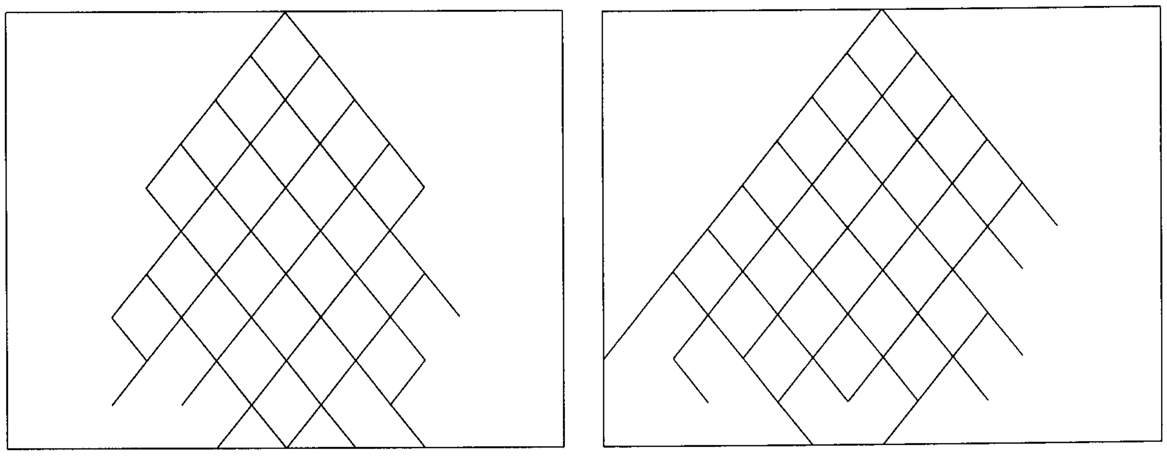


Figure D.2: Binary LZW trees of dictionary entries formed when encoding (a) the TeX file with normal LZW, (b) the TeX file with homophonic LZW, (c) the English text file with normal LZW, and (d) the English text file with homophonic LZW



(a) (b)

Figure D.3: Binary LZW trees of dictionary entries formed when encoding (a) the e-commerce data with normal LZW and (b) the e-commerce data with homophonic LZW

tree, and the tree on the right hand side show the homophonic encoded tree.

D.2 Statistical Results

The next figures illustrate the statistical results of the files. Each file is compressed with the binary ICL LZW algorithm and encoded with the homophonic ICL LZW algorithm. Fig. D.4 shows the results obtained for the html file. Fig. D.5 shows the results obtained for the TEX file. Fig. D.6 shows the results obtained for the English literature text file. Finally, Fig. D.7 shows the results obtained for the e-commerce data.

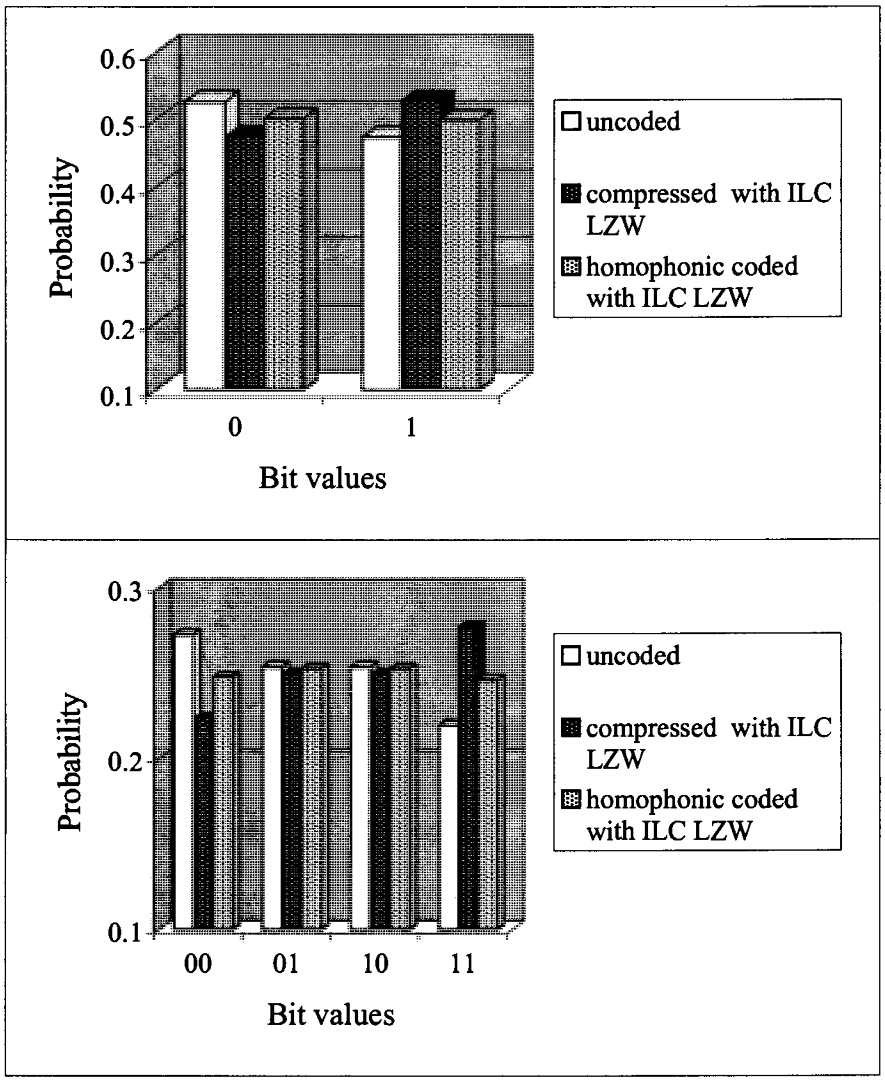


Figure D.4: Statistics of the uncoded and encoded html file

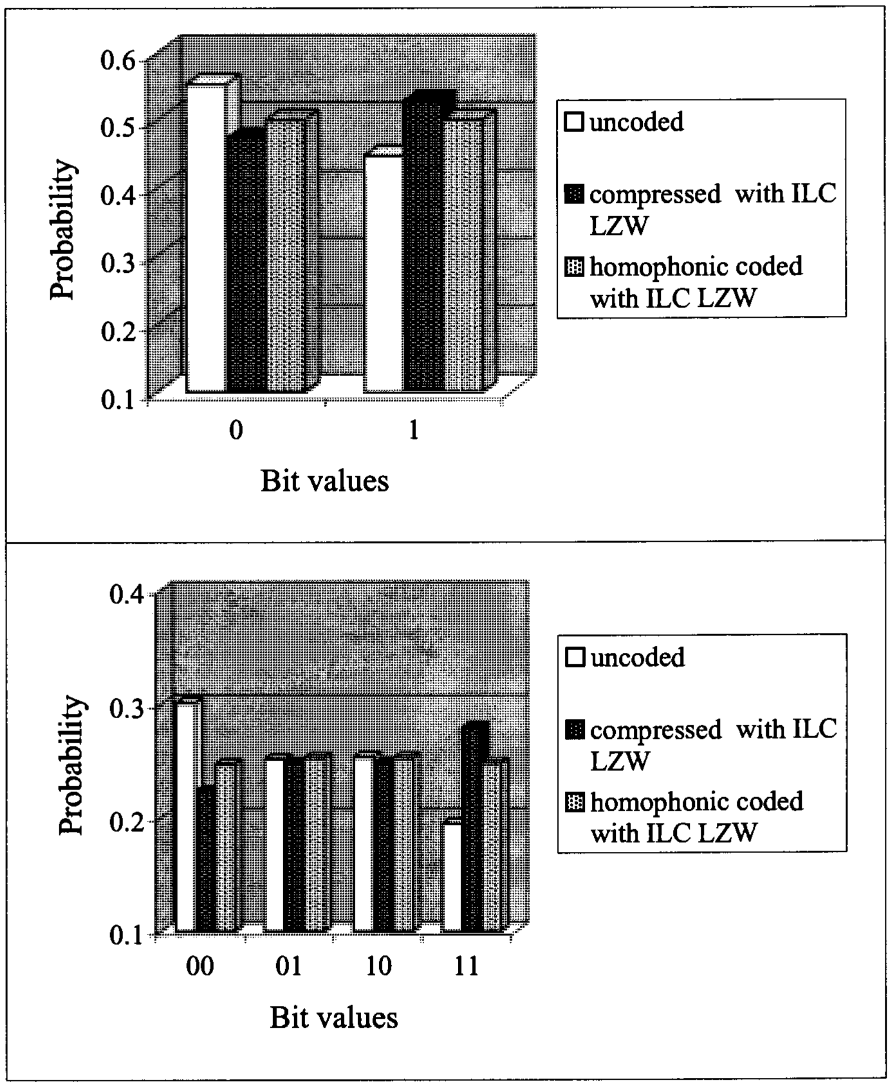


Figure D.5: Statistics of the uncoded and encoded TEX file

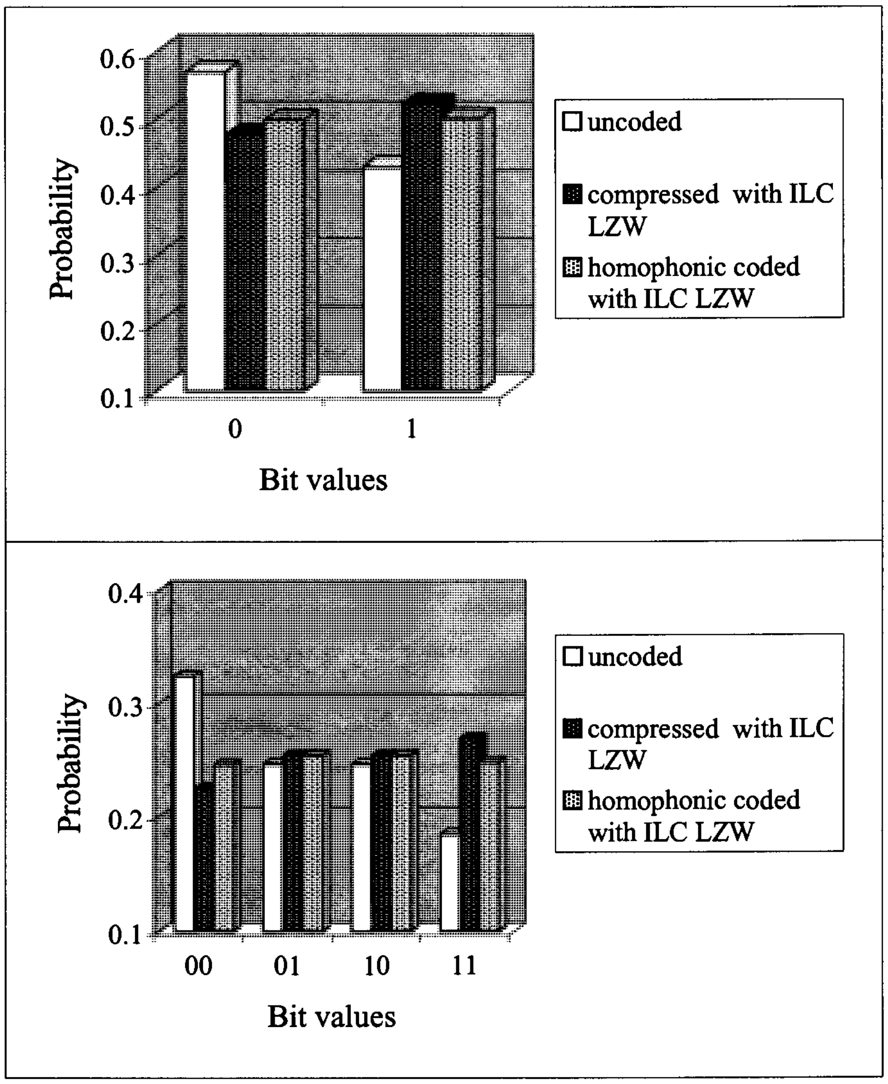


Figure D.6: Statistics of the uncoded and encoded English text file

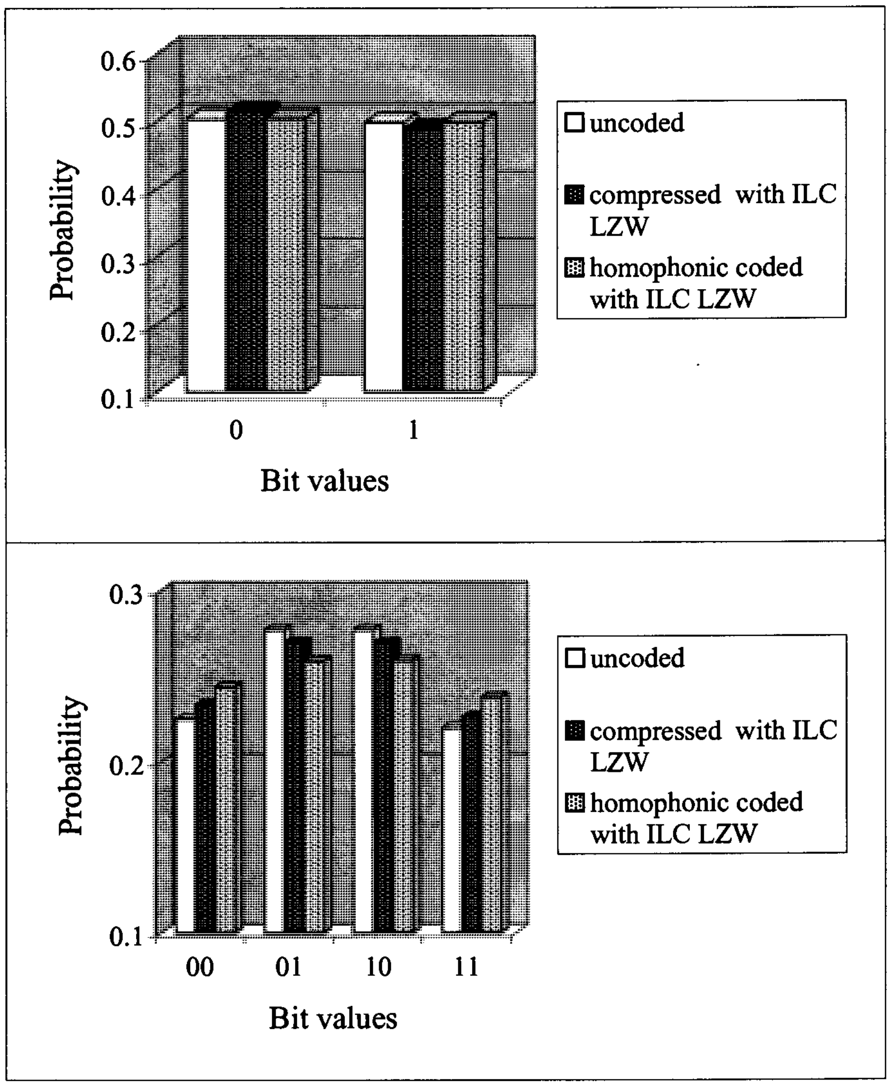


Figure D.7: Statistics of the uncoded and encoded e-commerce data

D.3 Entropy Results

Fig. D.8 to Fig. D.11 shows the binary entropy plots for the various files.

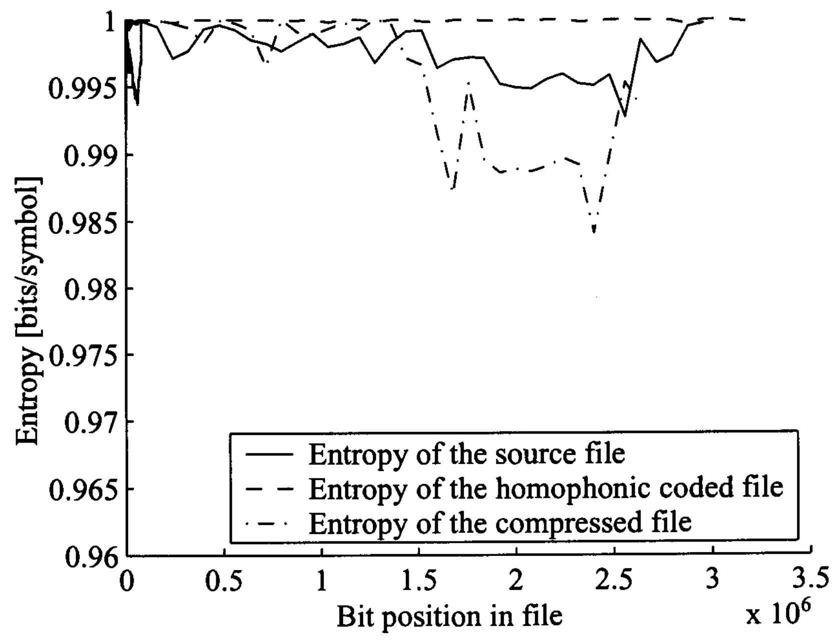


Figure D.8: Binary entropy of the uncoded, compressed and homophonic encoded html file

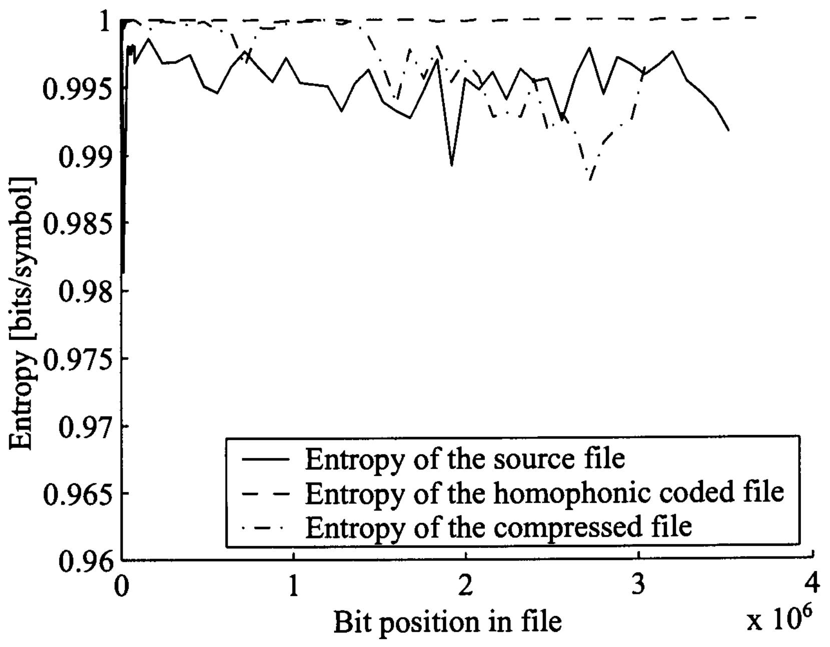


Figure D.9: Binary entropy of the uncoded, compressed and homophonic encoded TEX file

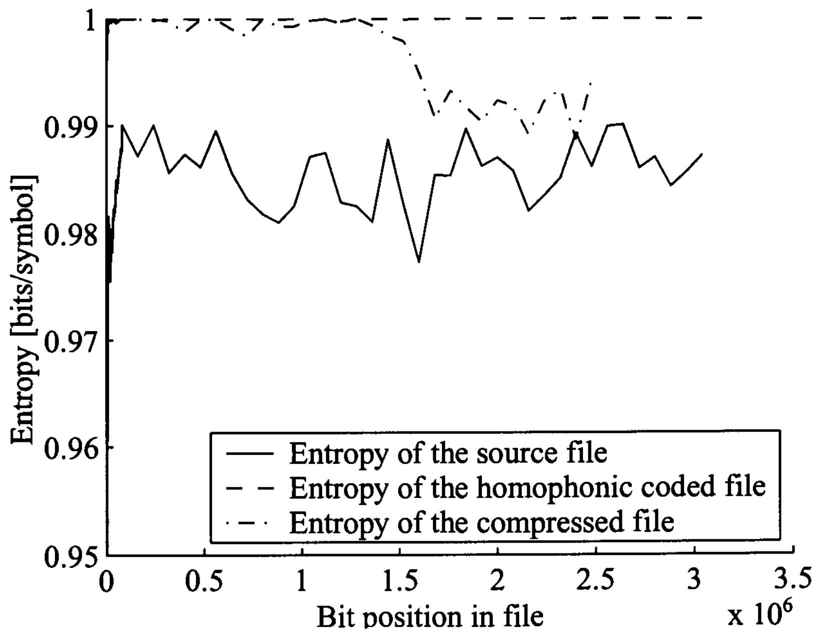


Figure D.10: Binary entropy of the uncoded, compressed and homophonic encoded English text file

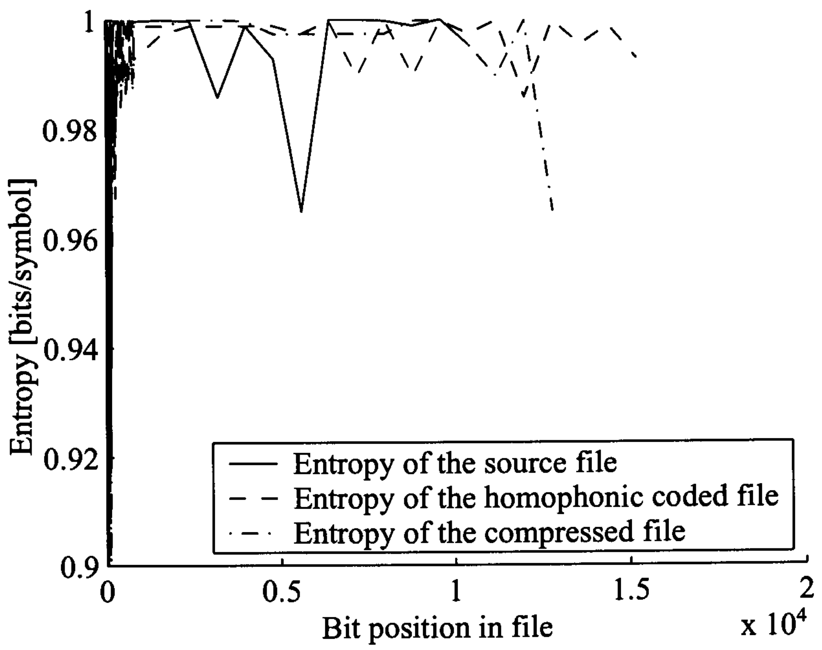


Figure D.11: Binary entropy of the uncoded, compressed and homophonic encoded e-commerce data

D.4 Bits/Symbol Results

Fig. D.12 to Fig. D.15 shows the bits/symbol results for the rest of the files.

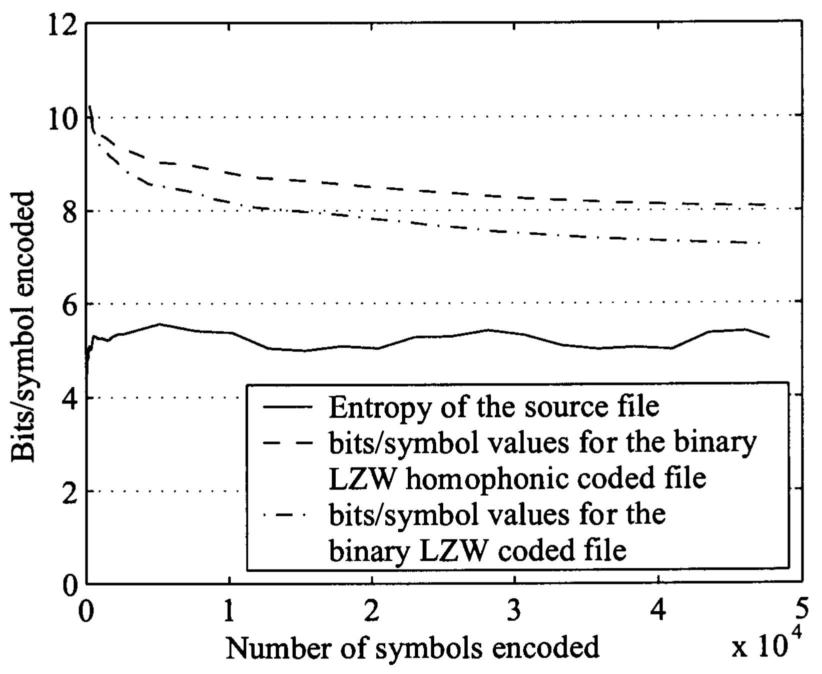


Figure D.12: bits per symbol plotted against number of symbols encoded for the compressed and homophonic encoded html file

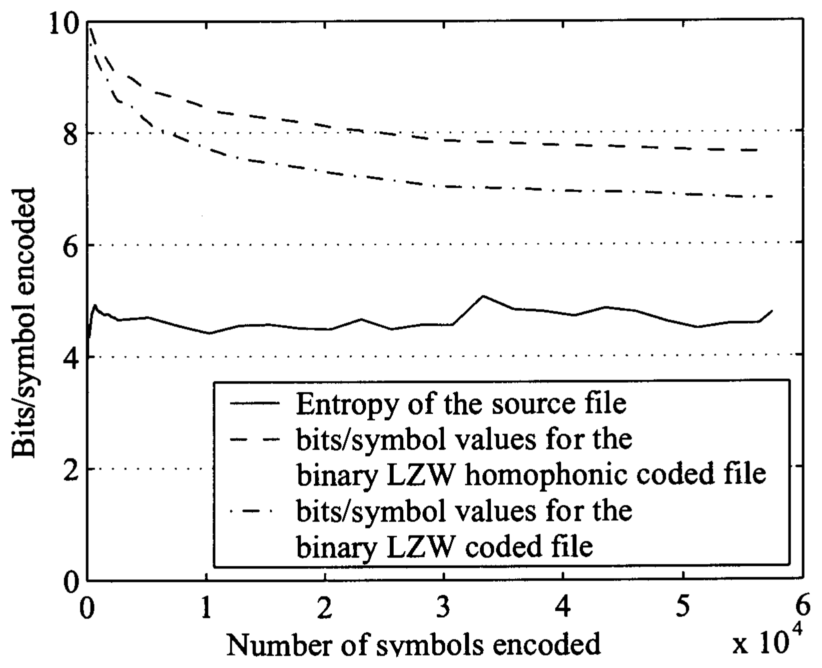


Figure D.13: Bits per symbol plotted against number of symbols encoded for the compressed and homophonic encoded TEX file

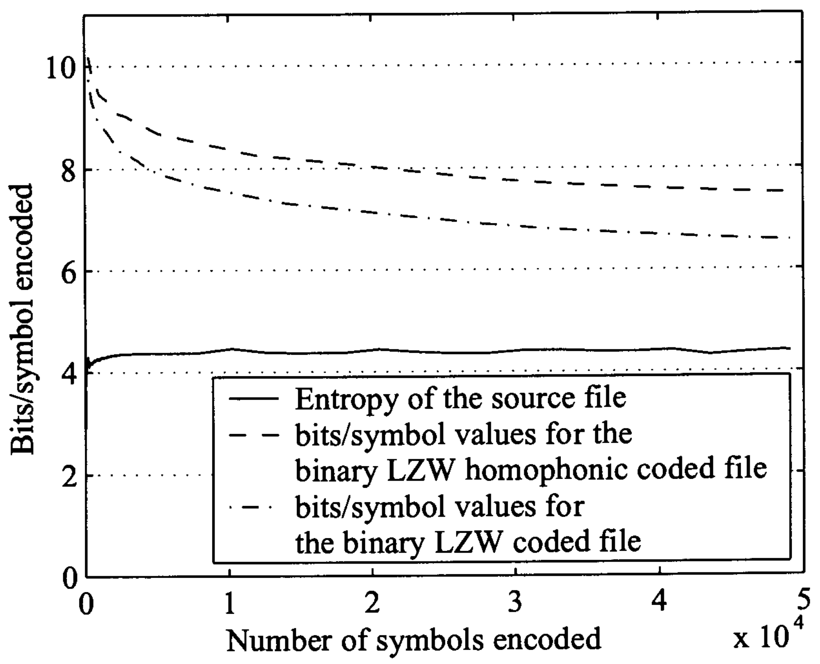


Figure D.14: Bits per symbol plotted against number of symbols encoded for the compressed and homophonic encoded English text file

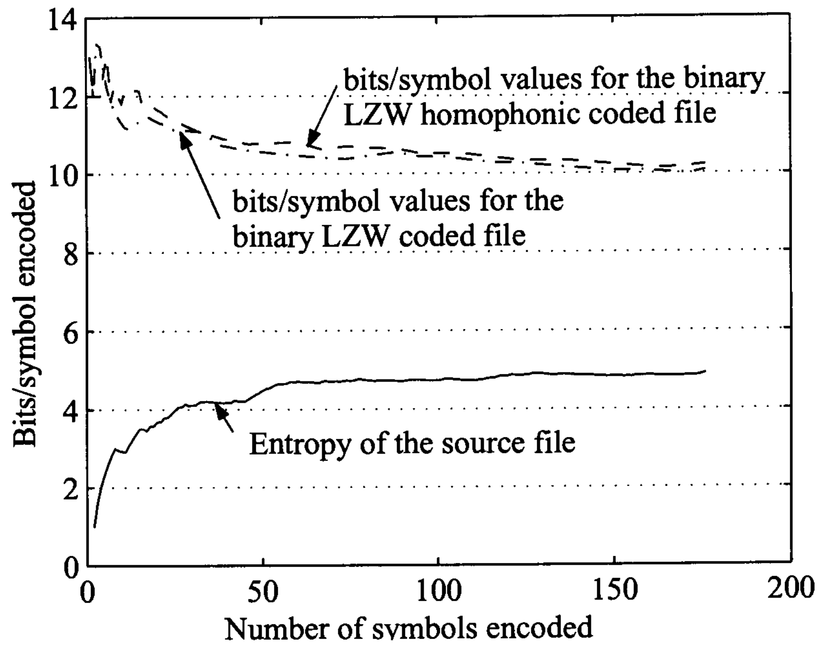


Figure D.15: Bits per symbol plotted against number of symbols encoded for the compressed and homophonic encoded e-commerce data

EXAMPLE OF ADAPTIVE MODEL UPDATING IN ARITHMETIC CODING

This example is intended to illustrate the model updating procedure that was used in the adaptive homophonic arithmetic coding program.

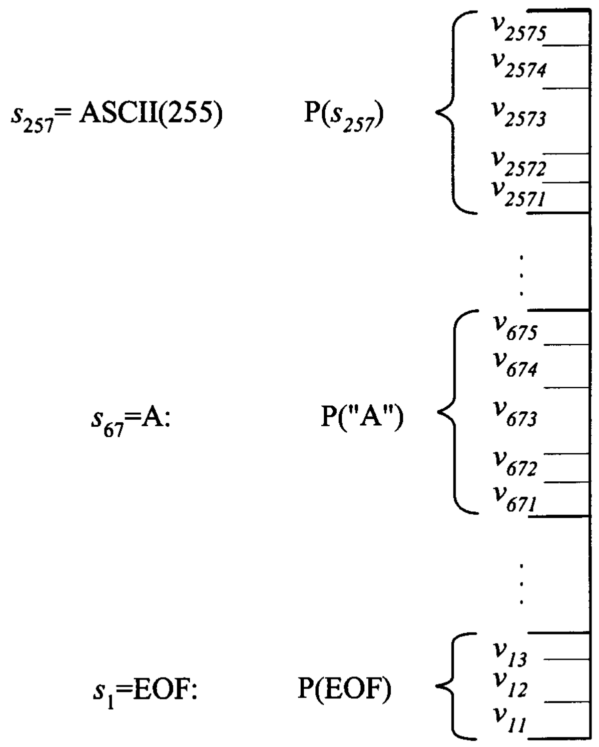


Figure E.1: Example of symbol order assignment to subintervals

Example:

Say that a character read in some time during encoding is an A and the output of its corresponding IIR filter is 0.9. Suppose the value of T_k is 2.71 at that particular time. First, randomly choose and encode one of A's current homophones in the `symbols` array (by making use of the `SelectHomphone` subroutine). Then multiply all values in the `charwidths` array and `Tk` with α, and add 1 to `Tk` and `charwdths[A]`, which then becomes $0.9 \times \alpha + 1$. Say $\alpha = 0.9$ so that A's weight is now 1.81 and `Tk`=3.439, and $q = 1.53 \times 10^{-4}$. Now run through all symbols in the model and calculate their new homophones' cumulative probabilities and lengths. Assume that the order in which the input symbols are assigned to a subinterval is `EOF` first, then the ASCII character for 0,1,2,..up to 255, which fits in the top couple of subintervals, as shown in Fig. E.1. The `EOF` character's weight is q, giving it a probability of 10^{-16}, which is 0000000000000001, so the first homophone is:

Homophone number hom	Homophone H	$P(H) = 2^{-i}$	i
1	0000000000000001	1/65535	16

The first entry in the `Symbols` structure array will be:

hom	`Symbols[hom].char`	`Symbols[hom].CumFreq`	`Symbols[hom].Length`
1	`EOF`	0000000000000000	16

The weight of A is 1.81, resulting in a probability of occurrence of

$$\frac{1.81 + 1.53 \times 10^{-4}}{3.439 + 257(1.53 \times 10^{-4})} = 0.5204,$$

or 1000010100111001, as calculated by the "division" process, described in Section A.2 on Page 3. A's homophones will thus be:

Homophone H	$P(H) = 2^{-i}$	i
1000000000000000	1/2	1
0000010000000000	1/56	6
0000000100000000	1/256	8
0000000000100000	1/2048	11
0000000000010000	1/4096	12
0000000000001000	1/8192	13
0000000000000001	1/65536	16

This table shows that the number to shift the buffer when adding the next codeword is equal to the round number of the homophone calculation loop i. Assuming that all the characters appearing in the character stream have ASCII characters higher than 65 ("A"), that is, the characters 0-64 all have frequency counts of q, the corresponding entries in the `Symbols` structure array, and thus codewords for A, are:

`hom`	`Symbols[hom].char`	`Symbols[hom].CumFreq`	`Symbols[hom].Length`
67	A	0000000001000011	1
68	A	1000000001000011	6
69	A	1000010001000011	8
70	A	1000010101000011	11
71	A	1000010101100011	12
72	A	1000010101110011	13
73	A	1000010101111011	16

The process continues for all characters in the adaptive model.

APPENDIX F

EXAMPLE OF DECODING OF THE HOMOPHONIC LZW ALGORITHM

This Appendix illustrates an example of the decoding procedure for the homophonic LZW algorithm, as described in Section 5.3. The same binary input sequence is used as was used in the example of Section 5.3, i.e. 1001101101. The random sequence that was used for the randomization was 101010. Note that, although not shown, the bits of this random sequence are received after every codeword. An *xor* operation has to be performed every decoding step to undo the randomization of the encoder input. Let i be the number of the decoding round and r_i the ith random bit in the total random sequence 101010. The randomization of the encoder input is undone in the following manner: The variable `randombit`, as used in the decoding program described in Section A.3.2, is also assigned a subscript i to indicate the value of `randombit` used in the ith round. This value is then given by

$$\texttt{randombit}_i = \texttt{randombit}_{i-1} \oplus r_{i-2} \qquad i = 1, 2, 3 \ldots . \tag{F.1}$$

$\texttt{randombit}_0$ is initialized with 0, whilst r_0 and r_{-1} are both assigned the value of 0 so that the first two output sequences are output unchanged. (This is necessary because the encoding algorithm starts by parsing off the first two bits and begins randomization on the third bit). Let the normal output sequence at any stage of decoding be $b_1 b_2 b_3 \ldots$ up to the number of bits in the particular output sequence. b_1 is the most significant bit of that particular output sequence. The randomization at stage i is undone by *xor*ing the particular b_1 with $\texttt{randombit}_i$ and the remainder of the sequence $b_2 b_3 \ldots$ with $\texttt{randombit}_{i+1}$. Table F.1

illustrates the process and column 6 shows the result of this *xor* operation - the actual output.

TABLE F.1: **Decoding of the sequence 1001101101 encoded with the homophonic LZW algorithm**

Round	r_i	`randombit`$_i$	Received	Normal	*xor*ed	Dict. entry		
i			Codeword	Output	Output	#	p	IC
1	r_1=1	`randombit`$_1$=0	1	1	1	3	1	**0**
2	r_2=0	`randombit`$_2$=0	0	0	0	4	0	**1**
3	r_3=1	`randombit`$_3$=1	3	10	01	5	3	**0**
4	r_4=0	`randombit`$_4$=1	0	0	1	6	0	**0**
5	r_5=1	`randombit`$_5$=0	4	01	01	7	4	**1**
6	r_6=0	`randombit`$_6$=0	1	1	1	8	1	**1**
7		`randombit`$_7$=1	3	10	01			
		`randombit`$_8$=1						

APPENDIX G

PART OF A LOG FILE

This appendix illustrates part of a log file that was kept during the homophonic arithmetic encoding simulations. The particular file being encoded in this case is a `C++` source file. The purpose of the log file entries is to illustrate the actual steps performed by the algorithm. The entries in the log file is simply the buffer values (in binary) calculated during encoding of the file and the symbol being encoded at that stage. Tabs are inserted to indicate where the 16 most significant bits are written to disk (when the buffer bit length exceeds 48).

```
0010001011011101 #
0010001101011100011010000 i
0010001101011100011010001111110110110 n
0010001101011100011010001111110110110101010101100111 c
                011010001111110110110101010101100111
                01101000111111011011010101010101101000011111101010011 l
                                1101101010101010110100001111101010011
                                110110101010101011010000111111010101001111111011100110 u
                                                11010000111111010101001111111011100110

110100001111110101010011111110111001101010101011 01001 d
                01010011111110111001101010101011 01001
                0101001111111011100110101010101101001110110110100100 e
                                001101010101011010011101101101 00100
                                0011010101010110100111011011010010000000000001000000
                                                100111011011010010000000000001000000

100111011011010010000000000010000000111000111 01001 <
                10000000000100000001110001110 1001
                1000000000010000000111000111010101111111011100110 v
                10000000000100000001110001110101011111110111001101000110000001111 c
                                0001110001110101011111110111001101000110000001111
                                000111000111010101111111011100110101000010110001110 0 l
```

```
                                                011111101110011010100001011000111000

0111111011100110101000011011001001001100 .
01111110111001101010000110110010010011100100011111110 01 h
                1010000110110010010011100100011111111001
                101000011011001001001110010001111111001100010000110111 >
                                01001110010001111111001100010000110111
                                0100111001000111111100110001000011011100000000000011010
                                                1110011000100001101110000000000011010

1110011000100001101110000000000001101000000000000010100

                1011100000000000110100000000000010100
                101110000000000011010000000000001010001100101010111 1 #
                                101000000000000101000110110001111111100001 p
                                1010000000000001010001101100011111111000101100101011110 01 r
                                                100011011000111111100010110010101111001

1000110110001111111000101100101011110011100001100101 00 a
                111000101100101011110011100001100101 00
                1110001011001010111100111000011001010100101101110 1000 g
                                111100111000011001010100101101110 1000
                                11110011100001100101010010110111010011000111110011100 m
                                                0101010010110111010011000111110011100

010101001011011101001100011110011100101110101101011 a
                0100110001111001110010111010110101 1
                01001100011110011100101110110100100001 1001
                0100110001111001110010111011010011010100110100110 h
                                1100101110110100110101001101001 10

1100101110110100110101101011111101101 0 d
110010111011010011010110110110111101110100 0 r
11001011101101001101011011011100110010100110000 0 s
1100101110110100110101101101110011001010011000011101101111110111 t
                110101101101110011001010011000011101101111110111
                110101101101110011001010011000011101101111110001000101010110 11 o
                                11001010011000011101101111110001000101010110 11

11001010011000011101101111110001000101010111001001011001010 01 p
                1101101111110001000101010111001001011001010 01
                11011011111100010001010101110010010110011100010011 00
                                000101010111001001011001110001001 100
                                00010101011100100101100111000100110001010 0
```

APPENDIX H

A LINEAR AGING MODELLING METHOD

H.1 Introduction

This appendix describes the method to perform linear aging probability estimation, based on the sliding window statistical modelling method, as was used in the experiments.

H.2 Linear aging probability estimation

Adaptive probability estimation models utilize a string of previously encountered symbols to estimate the statistics of symbols to come. There are two different ways to consider the length of this string of characters. The first method is to count the characters from the beginning of the stream, as illustrated in Fig. H.1, and the second method is to count the characters only from the window of n past encountered characters, as illustrated in Fig. H.2. The first method gives a global estimate of probabilities, where the second method gives a local estimate of probabilities. True adaptive models are supposed to adapt to changes in source statistics. Although both methods will achieve this, the second method will adapt more rapidly. To calculate accurate probabilities, the length of the window n must be at least 10 times the number of possible characters that can appear in the input stream. Since the files used in the experiments were considered to be 8-bit character sources, it means

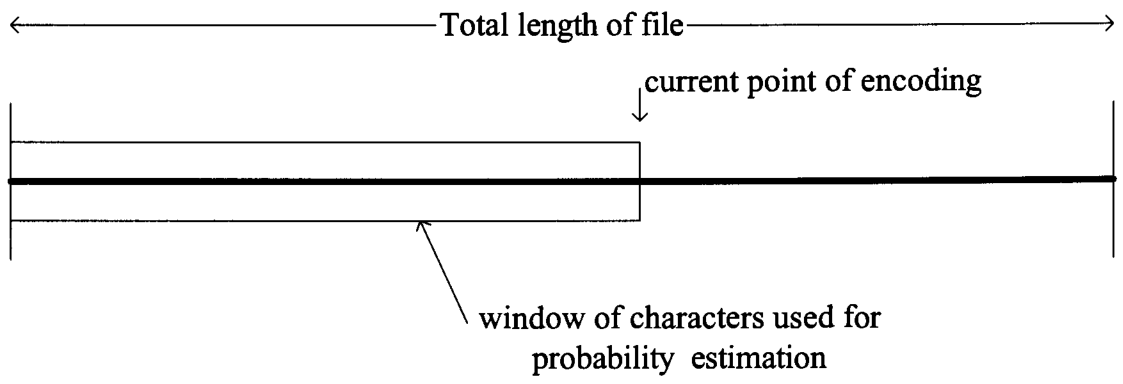

Total length of file
current point of encoding
window of characters used for probability estimation

Figure H.1: Adaptive probability estimation by counting all characters encountered since the beginning of the file

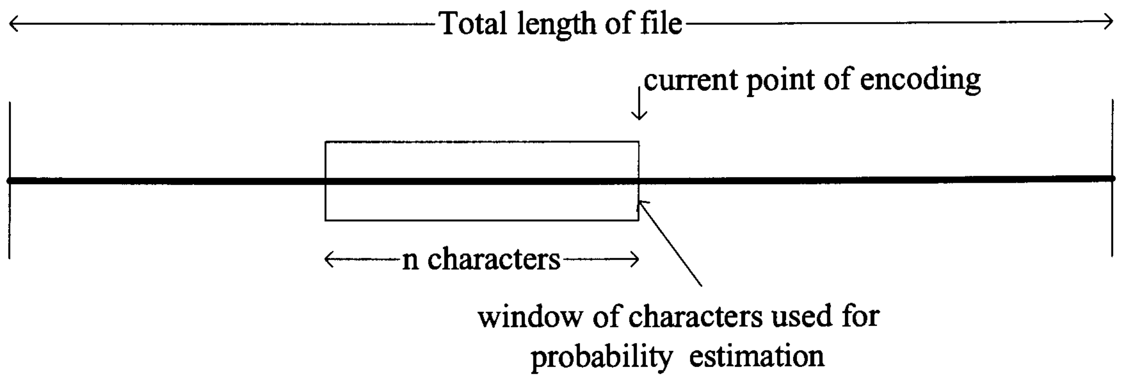

Total length of file
current point of encoding
n characters
window of characters used for probability estimation

Figure H.2: Adaptive probability estimation by counting characters in the window of n past encountered characters

that the past $10 \times 2^8 = 2560$ characters in the stream are utilized to predict the statistics of characters to come.

This sliding window technique introduces a *locality of recency* effect [22]. It considers the past n processed characters to be equally important, and all characters that appeared before that not important at all. This is a rather coarse method of assigning recency when encoding. A better method is to consider characters near the end of the window to be more important than characters in the beginning of the window when the probabilities are estimated (just like the IIR method does). For linear aging models, it can be achieved in the following manner: Each character in the window is assigned a weight y. This value depends on the position of the character in the window, and is given by

$$y = \lfloor mx + c \rfloor, \tag{H.1}$$

which is the formula for a straight line, where
x is the position of the character in the window, and ranges between 1 and 2560
c is the minimum value that y can be assigned,
m is the slope of the line, which has to be determined, and
$\lfloor X \rfloor$ is the largest integer smaller than X.

The frequency counts of the characters are stored in 16 bit unsigned integers in the program, so the total of all 2560 calculated weights may not exceed $2^{16} = 65536$, and the minimum frequency count that can be assigned to a character is 1, corresponding to a probability of 1/65536. The second limitation indicates that the value of c must thus always be 1. The first limitation is used to calculate the value of m in the following manner: Note from Fig. H.3 that the line described by $y = mx + c$ goes through the average value of the respectively calculated weights in the middle of the window (if the window is filled up). At that specific point, the value of x is 2560/2=1280, the value of y is the average of the weights, 65536/2560 = 25.6, and since $c = 1$, m can be determined from Equation (H.1). The probability of each input character is given by summing all the weights assigned to the character and taking the ratio of the result to 65536.

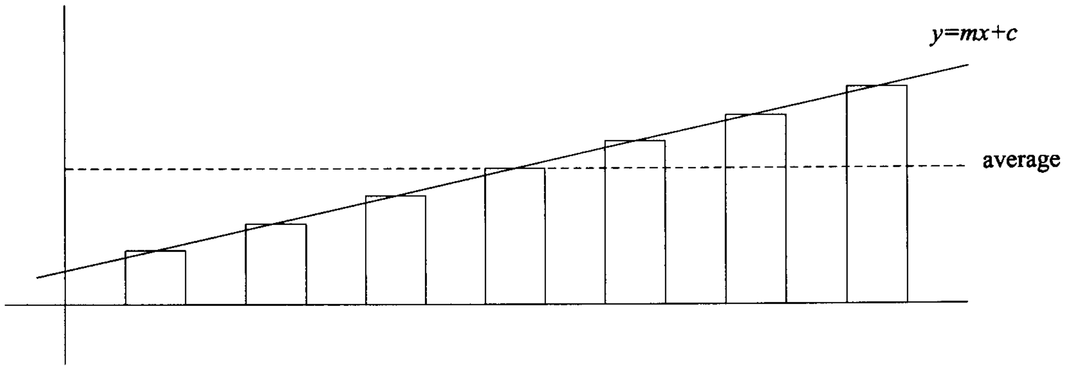


Figure H.3: Determination of m

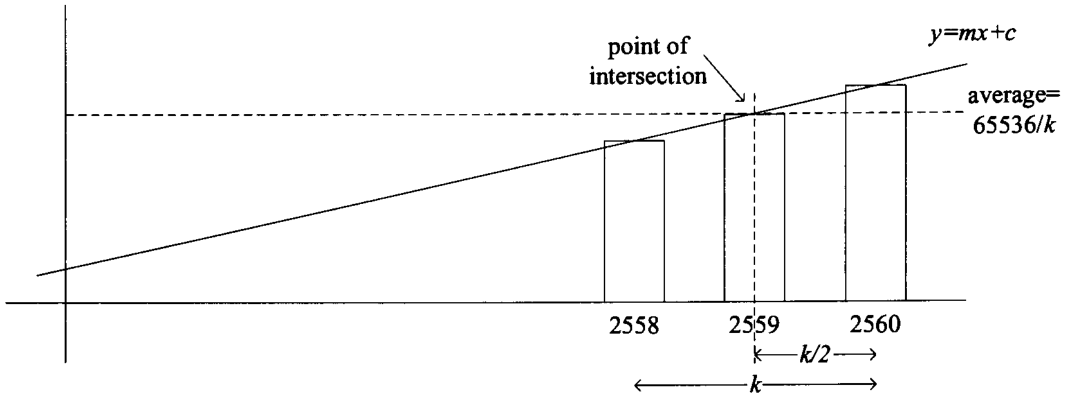


Figure H.4: Determination of m before the first 2560 characters are encountered

At the start of encoding, the window is not filled up yet, and characters shifted into the window are shifted in from the right, where the values of x are large. In this case, the point where the straight line intersects the average value of the weights is not in the middle of the window, as illustrated in Fig. H.4. The value of x at this point is 2560-k/2, where k is the number of characters in the window. m is thus given by

$$m = \frac{65536/k - 1}{2560 - k/2}. \tag{H.2}$$

During this phase where the window is not yet filled up, the model is updated every time a symbol is encountered. Furthermore, an end-of-file symbol EOF have to be added to the model, to indicate to the decoder when decoding should terminate. The frequency count associated with this symbol is taken as 1, since it will only occur once - at the end of the sequence. Because all 257 characters must have a frequency count of at least 1, it is required that the total of all the frequency counts, N_T, may not exceed $N_T = 65536 - N_{cniw}$, where N_{cniw} is the number of characters that did not appear in the window. This value must be calculated each time before the window is shifted. The actual formula used to calculate m is thus

$$m = \frac{N_T/k - 1}{2560 - k/2}. \tag{H.3}$$

When the window is filled up, and every time it is shifted, the model is updated by calculating the new character frequency counts.