UNIVERSAL HOMOPHONIC CODING

by

CHARLES CATER STEVENS

submitted in part fulfilment of the requirements
for the degree of

MASTER OF SCIENCE

in the subject

COMPUTER SCIENCE

at the

UNIVERSITY OF SOUTH AFRICA

SUPERVISOR: DR W B SMUTS

NOVEMBER 1996

Abstract

Redundancy in plaintext is a fertile source of attack in any encryption system. Compression before encryption reduces the redundancy in the plaintext, but this does not make a cipher more secure. The cipher text is still susceptible to known-plaintext and chosen-plaintext attacks.

The aim of homophonic coding is to convert a plaintext source into a random sequence by randomly mapping each source symbol into one of a set of homophones. Each homophone is then encoded by a source coder after which it can be encrypted with a cryptographic system. The security of homophonic coding falls into the class of unconditionally secure ciphers.

The main advantage of homophonic coding over pure source coding is that it provides security both against known-plaintext and chosen-plaintext attacks, whereas source coding merely protects against a ciphertext-only attack.

The aim of this dissertation is to investigate the implementation of an adaptive homophonic coder based on an arithmetic coder. This type of homophonic coding is termed universal, as it is not dependent on the source statistics.

Keywords: arithmetic coding; adaptive models; cryptography; data compression; homophonic coding; homophonic substitution; modelling; randomness; statistical testing; secrecy; unconditional security.

TABLE OF CONTENTS

LIST OF TABLES AND FIGURES

Chapter 1

INTRODUCTION

1.1 Background

The security of a large number of cryptographic systems is based on the computational complexity of certain mathematical algorithms. Unconditionally secure systems are preferable [Günther, p.406]. These are cryptographic systems which are not dependent on computational complexity issues or limited to cipher-text only attacks [Rueppel, p.73]. This can be achieved by conditioning the plaintext (the message before encryption) by reducing the redundancy with data compression and increasing the entropy by means of a randomising process.

Redundancy in plaintext is a fertile source of attack in any encryption system. Compression before encryption reduces the redundancy in the plaintext message, but it has been shown that this does not make a cipher more secure [Boyd 1992, p.10], [Irvine]. The cipher text is still susceptible to a known plaintext attack.

The aim of homophonic coding is to convert a plaintext source into a completely random sequence. It has been shown [Jendal, Kuhn & Massey, p.383] that when a random sequence is encrypted by a non-expanding cipher, then the ciphertext is completely random and independent of the key. This is achieved by randomly mapping each source symbol into one of a set of homophones. Each homophone is then encoded by a source coder after which it can then be encrypted with a conventional cryptographic system.

The main advantage of homophonic coding over pure source coding is that it provides security both against known-plaintext and chosen-plaintext attacks [Penzhorn]. On the other hand, source coding merely protects against a ciphertext-only attack.

The aim of this report is to investigate the implementation of an adaptive homophonic coder based on an arithmetic coder. This type of homophonic coder is termed universal as it is not dependent on the source statistics of the message.

1.2 Statement of the Problem

Homophonic coding based on a *fixed model* of the message source, (the source statistics used in the model are fixed) does not provide optimal results when used to encode a source with varying statistics.

1.3 Hypothesis

The hypothesis is made that arithmetic coding can be used in combination with homophonic coding to implement a homophonic coder which can adapt to source statistics. This hypothesis has been proposed by Penzhorn [Penzhorn]. However, no record of any implementation could be found.

A second hypothesis is that no special form of homophonic arithmetic decoding is required. The arithmetic decoder will follow the homophonic coder without having to know anything about the homophones.

1.4 Objectives of this Study

1. To gain an understanding of data compression.
2. To gain an understanding of cryptography.
3. To gain an understanding of homophonic coding.
4. To investigate the topic of randomness and tests for detecting non randomness in a sequence.
5. To implement a universal homophonic coder.
6. To evaluate the results from the implementation.
7. To state the difference between homophonic substitution and homophonic coding.

1.5 Scope of this Investigation

The scope of this dissertation is the implementation of universal homophonic coding. We consider two forms of homophonic coding: one based on a conventional methods which merely expands the source alphabet and a second form based on dyadic[1] homophones. The real test of the theory would be to

[1] Homophones with probabilities based on negative integer powers of two (binary powers).

subject the output of the homophonic coder to cryptanalytic attacks. These forms of attack are not covered in this research.

The order of the adaptive model will be limited so that it can easily fit into the memory available. Optimisation and speed do not receive direct attention in this dissertation.

1.6 Relevance of the Research

Compression before encryption does not increase the security as the system is still weak against a chosen-ciphertext attack [Bergen & Hogan 1993]. This is especially true with redundant sources such as English text. Boyd [Boyd 1991, p.275] investigated homophonic substitution and source coding and tabled a question whether homophonic coding had any advantages over conventional data compression. This research will try to justify homophonic coding in conjunction with an arithmetic coder.

An important aspect of the form of homophonic coding together with an encryption system is that it provides security against both known-plaintext and chosen-plaintext attacks [Penzhorn, p.343].

1.7 Previous Work

The concept of homophonic coding is not new. [Massey] and [Günther] have introduced the ideas of variable length homophonic coding. [Penzhorn] introduced the use of a fast shift and add arithmetic coder to the homophonic interval based on a static model.

1.8 Methodology

The following methodology was employed in conducting the research for this dissertation.

- Initially, an in depth study was made into the subject of lossless data compression [Stevens 1992]. The result of this study was the development of V.42 bis data compression for commercial use in modems[2].

- Cryptography was studied and the investigation of "randomness" was undertaken [Stevens 1994]. The results of this randomness investigation and the necessary code to perform randomness tests

[2] Duxbury 996 and Bell Atlantic managed modem range.

were released on the Internet newsgroup *sci.crypt*. This code is now available at major cryptographic Internet sites[3].

- A prototype system based on an adaptive model was implemented.
- Tests were carried out on the output of the prototype.
- The results were evaluated.

1.9 Organisation of this Document

Universal homophonic coding comprises of a number of distinct elements. For this reason, this document is divided into a number of sections. These sections are data compression, cryptography and homophonic coding.

Chapter 2 deals with the meaning and measuring of "randomness" to ensure the perfect cipher. This chapter is complex and may be skipped when initially reading this dissertation.

Chapters 3 and 6 are basic introductions to the subjects of data compression and cryptography respectively. Chapter 3 reviews entropy and arithmetic coding that are fundamental requirements in later chapters. Furthermore, it shows how redundancy is removed and introduces the reader to the topic of modelling.

Chapter 4 describes arithmetic coding in further detail. Arithmetic coding is the key concept in the form of homophonic coding discussed in this dissertation. Modelling the input source is a key concept in any form of data compression system. This is discussed in chapter 5.

Chapter 6 reviews cryptography in which the subject of a classical homophonic cipher is introduced. Other forms of ciphers are also discussed. The information theory of cipher security is discussed for later analysis of homophonic coding.

Chapters 7 and 8 consider homophonic coding in detail. Chapter 7 introduces the subject and shows the state of the current published research. Chapter 8 discusses the form of homophonic coding used in this dissertation.

[3] University of Milan, Italy: ftp.unimi.dsi.it in /pub/security/cryptanalysis/alltests.tar.zip and University of Oxford, U.K.: ftp.ox.ac.uk in /pub/cryptanalysis/alltests.tar.gz

Chapter 9 discusses the implementation of the universal homophonic coder and finally Chapter 10 reviews the results.

All the necessary source code is to be found in the appendix. A glossary of terms is given in an appendix.

Chapter 2

RANDOMNESS

2.1 Introduction

This chapter reviews the controversial topic of randomness. The output of the homophonic coder must pass some of the standard statistical tests. These tests will detect non randomness in a sequence. There is no single test that can prove randomness. Testing will only increase the confidence in the security of the system.

Randomness is used in numerous fields such as cryptography, simulation, computational number theory, computational geometry, VLSI testing, parallel and distributed computing. This chapter addresses randomness for cryptographic purposes.

In the field of cryptography, unpredictability is a key criterion. The sequence must be unpredictable to the left and also unpredictable to the right. This means that given any portion of the sequence, it must be impossible to determine either the past sequence or to determine the next portion of the sequence.

Cryptanalysis uncovers either a structural or statistical weakness in a cryptographic algorithm. In theory, every cipher can be broken once it exceeds its unicity distance (this term will be explained in a later section), e.g., by an exhaustive key (brute force) search. A major design goal of a cipher system is to make it secure against every feasible attack.

2.2 What is Randomness?

Maurer defines *randomness "as a property of an abstract mathematical model characterised by probabilities"* [Maurer, p.90]. Whether this model can give an exact description is a philosophical question.

Randomness is said to be endemic in many physical processes such as shot noise or even quantum mechanics. This is known as true or abstract randomness. Pseudo random sources try to imitate true random sources.

In this study, pseudo-random sequences are the main focus for three main reasons.

1. A pseudo-random bit generator produces from a short seed, a longer sequence of pseudo-random bits from a deterministic algorithm.

2. The sequence is easily reproducible given the knowledge of the seed.

3. They are extensively used in cryptography for key sources.

Randomness is related to the ability to predict the next outcome in a sequence [Feder, Gerhav & Gutman, p.5]. True random sources can be considered unconditionally unpredictable. If the outcome can be predicted, then the sequence cannot be considered random. The problem of predicting the next bit in a sequence has intrigued information theorists for many years. The complexity of the source is related to the amount of information in it (measured by the entropy) and how well it can be compressed. A sequence is totally unpredictable if and only if it is incompressible [Feder, Merhav & Gutman, p.9].

The following section will give some of the many definitions and notions of randomness.

2.2.1 Intuitive Definitions

Intuitively, the probability of predicting the next bit of a random sequence is ½ as in the coin tossing experiment. Given full knowledge of all preceding bits, is it computationally infeasible to predict the next bit with better than 50% accuracy. Each bit of the sequence should be unpredictable and the sequence must pass several statistical tests [Knuth].

2.2.2 Kolmogorov Definition

Kolmogorov showed that a string of bits is random if it cannot be described by a string shorter than itself. The Kolmogorov complexity[4] K, of a binary string x is defined as the length l of the shortest program p to print out this string on a universal computer U. [Cover & Thomas, p.147].

$$K_U(x) = \min_{p:U(p)=x} l(p)$$

If the string is random, the Kolmogorov complexity is close to the entropy [Cover & Thomas]. This implies that the string cannot be compressed, i.e. there is no redundancy assuming perfect compression.

A sequence $x_1, x_2, \ldots, x_n$ is said to be *algorithmically random* [Cover & Thomas, p.157] if: $K(x_1 x_2 \ldots x_n | n) \geq n$

An infinite string x is *incompressible* [Cover & Thomas, p.157] if: $\lim_{n \to \infty} \frac{K(x_1 x_2 x_3 \ldots x_n | n)}{n} = 1$

If a string $x_1, x_2, \ldots, x_n$ is incompressible, then it satisfies the law of large numbers, so that: $\frac{1}{n}\sum_{i=1}^{n} x_i \approx \frac{1}{2}$

Therefore, the number of ones and zeros in an incompressible string (and thus a random sequence) is almost equal. A proof of this fact[5] is given by [Cover & Thomas, p.157].

Although the Kolmogorov complexity is interesting for its definition of randomness and compressibility, the function is not computable. Therefore, it does not provide a practical test for randomness.

2.2.3 Complexity Definition

The randomness of an event is relative to a specific model of computation with a specified amount of computing resources [Blum & Micali]. This definition is not considered in this report.

2.2.4 Golomb's Randomness Postulates

Golomb proposed three randomness postulates for a binary sequence of period *p*, i.e. a pseudo-random sequence such that $s_{m+p} = s_m$ [Beker & Piper, p.169].

R1. *If p is even then the length of p shall contain an equal number of ones and zeros. If p is odd, the number of zeros shall be one more or one less than the number of ones.*

R2. *In the cycle of length p, half the runs have length 1, a quarter have length 2, and an eighth have length 3 and, in general, for each i for which there are at least 2^{i+1} runs, 2^{-i} of the runs*

[4] The Kolmogorov complexity concept, was independently and almost simultaneously discovered by Kolmogorov, Solomnoff and Chaitin.

[5] In practice, the number of ones and zeroes has been found to be a strong indicator of non randomness [Stevens 1994].

have length i. Moreover, for each of these lengths, there are equally many gaps (runs of zeros) and blocks (runs of ones).

R3. *The out-of-phase correlation is a constant.*

Sequences satisfying these criteria are known as G-random or PN-sequences (PN is the term for pseudo noise).

2.2.5 Binary Symmetric Sources

Any form of a random bit generator is designed to output a sequence of statistically independent and symmetrically distributed random variables. A pseudo-random bit generator outputs a sequence so that it appears that it was generated by a binary symmetric source (BSS). This sequence is often used as a synonym for random [Maurer, p.90].

When this definition of *randomness* for a sample sequence is used, it is then possible to form a known distribution so that the acceptance and rejection regions can be specified.

2.3 Statistical Testing For Randomness

In this section, several standard tests for randomness are discussed. Statistical tests detect a possible statistical defect in a random bit generator, that is that statistical model describing the sequence deviates from a binary symmetric source. A test examines sequence of length N. [Knuth] is one of the definitive works for statistical testing of random number generators (RNG).

Tests can only highlight a particular bias in the binary sequence. They are not conclusive. No single test can prove randomness. Finding the bias is not easy as most deterministic (pseudo random[6]) generators are designed to pass the usual statistical tests. For example, the binary representation of π passes most statistical tests but is deterministic.

Tests fail for two reasons, a sequence may deviate from an expected outcome or it may be too close to the expected outcome. Sub sequences should also be tested for local randomness as sometimes the cryptanalyst will have some of the sequence.

[6] Pseudorandom distributions are those distributions which cannot be efficiently distinguished from the uniform distribution on strings of the same length [Goldreich & Krawczyk, p.114]

2.3.1 Analysing different distributions

This section discusses how the actual test results are analysed. The test result is compared with a known statistical model. Samples are tested if they are from the similar populations.

Data is continuous or binned (a finite number of categories). Continuous data converted into binned data by grouping entries into specified ranges of continuous variables. Binning involves the loss of information. The chi-square test is used on binned data and the Kolmogorov-Smirnov (K-S) test is generally used on continuous data as a function of a single variable.

The next section considers two methods, the chi-square and the Kolmogorov-Smirnov tests, to compare distributions. There are many other tests and to name but a few; Neyman-Pearson, Student-t, and Kullback.

2.3.1.1 Chi-square Test χ^2

The chi-square test χ^2 measures the goodness-of-fit between some known distribution and the distribution of the sample. Let o_i be the event observed at the i^{th} bin and e_i is the number expected according to some known distribution, then:

$$\chi^2 = \sum_i \frac{(o_i - e_i)^2}{e_i}$$

The chi-square probability function $Q(\chi^2 | v)$ is an incomplete gamma function [Press, Teukolsky & Vettering, p.621]. $Q(\chi^2 | v)$ is the probability that the sum of the squares of v random normal variable of zero mean will be greater than χ^2. A low value obtained from $Q(\chi^2 | v)$ suggests a significant difference in the distributions.

The degree of freedom v is usually the number of bins minus one. The expected value is close to the degree of freedom, i.e., $E[\chi^2] \approx v$. The χ^2 statistic can be looked up in tables for different significance levels. The significant level is the total probability of all the outcomes contained in the critical region.

The χ^2 test is used to decide between the *null hypothesis*, i.e., the observation does not fall in the critical region,

- H_0:{p(j):0 $\leq$ j $<$ m} *is* the probability distribution of X_0, X_1, ...X_{n-1}; and the *alternate hypothesis*,
- H_1:{p(j):0 $\leq$ j $<$ m} *is not* the probability distribution of X_0, X_1,...X_{n-1}

The null hypothesis is made assuming that the samples are from the same population and the difference has arisen purely by chance. If the probability of getting two samples is less than, say 5%, the hypothesis can be rejected.

There are two versions of the χ^2 test to test the significance levels. The χ^2 value based on the degrees of freedom is viewed with the percentage points and the random test results are based on the deviations [Knuth].

2.3.1.1.1 One-Tailed

Using a single significance level is not used in this report as the results may be too good, i.e. an equal number of ones and zeros (i.e. not only are large χ^2 values improbable, so are small values).

2.3.1.1.2 Two-Tailed

The significance levels at each end of the χ^2 distribution are tested. An example of the critical regions are:

0 — 1%	99 — 100%	Reject
1 — 5%	95 — 99%	Suspect
5 — 10%	90 — 95%	Almost suspect

2.3.1.2 Kolmogorov-Smirnov Test

The K-S measure is the maximum value of the absolute difference between two cumulative distribution functions. It is applicable to unbinned functions that are functions of a single independent variable [Press, Teukolsky & Vettering, p.620].

Large values of n will average out local non-random behaviour and therefore the data should be blocked. The K-S test should then be applied to the results of the blocks.

As in the χ^2 test, the same hypothesis testing is applicable to the K-S test.

The K-S test is mentioned here for completeness and it not used in any of the tests in this report.

2.3.2 Empirical Tests

Empirical tests [Knuth] require computation to evaluate the data. There are many different empirical tests that can be performed, a few being given below. Many of the tests are limited by processing time and memory requirements. The majority of the tests use the chi-squared distribution to detect a bias.

2.3.2.1 Frequency or Equi-distribution Test

The frequency test, tests the hypothesis that the sequence is uniformly distributed, i.e. the number of ones and zeros is approximately equal. This test is a good indicator on non randomness. Previous testing has shown [Stevens, 1994] that if a sequence fails this test, it will fail all other tests.

Some tests that can be done:

Normal distribution [Maurer] given by $f_{T_F} = \frac{2}{\sqrt{N}}\left(\sum_{i=0}^{N} x_i - \frac{N}{2}\right)$

For a normal distribution with a large N and a zero mean with variance 1, the rejection thresholds are |2.5...3.0|.

Chi-square χ^2

Testing the number of ones (n_1) and the number of zeros (n_0) in a sequence is approximately the same, i.e. $p(0) = p(1) = 0.5$:

$$\chi^2 = \sum_{i=0}^{1} \frac{(n_i - 0.5)^2}{0.5}$$

[Beker & Piper, p.171] and [Gustafson, Dawson & Caelli, p.122] give the following which can be derived from the above equation:

$$\chi^2 = \frac{(n_0 - n_1)^2}{N} \quad and\ v = 1$$

With one degree of freedom and 5% significance levels, $0.00393 < \chi^2 < 3.841$

Increase the bin size:

Choose d to be a multiple of the word size, e.g. 256. For each value of r ($0 \le r \le d$), count the number of times that $o_i = r$ for $0 \le i \le N$. Apply the χ^2 test with $v = d - 1$ and the probability of d^{-1}.

With d = 256, 255 degrees of freedom and 25% significance levels, the critical region bounds are $239.39 < \chi^2 < 269.88$

2.3.2.2 Serial Test

The serial test checks that the consecutive entries are distributed uniformly and independently. This ensures that the transition probabilities are similar suggesting that each bit be independent of its predecessor.

From [Beker & Piper, p.172] and also [Gustafson, Dawson & Caelli, p.122]: let n_{ij} be the number of consecutive runs, i.e. n_{00} is the number of 00 runs.

$$n_{00} + n_{01} = n_0 - 1,\ n_{10} + n_{11} = n_1 - 1 \text{ and } n_{00} = n_{01} = n_{10} = n_{11} \approx \frac{N-1}{4}$$

$$\chi^2 = \frac{4}{N-1}\sum_{i=0}^{1}\sum_{j=0}^{1}(n_{ij})^2 - \frac{2}{N}\sum_{i=0}^{1}(n_i)^2 + 1 \quad \text{with } \nu = 2$$

With two degrees of freedom and 5% significance levels, $0.1026 < \chi^2 < 5.991$

With three degrees of freedom, the standard chi-squared equation is

$$\chi^2 = \frac{4}{N}\sum_{i=0}^{1}\sum_{j=0}^{1}\left(n_{ij} - \frac{N}{4}\right)^2$$

Neither of these two distributions follow the restriction of [Knuth] that $n_{2i,2i+1}$ should be the order of the test.

2.3.2.3 Poker Test

The poker test counts the number of similar patterns in a block of chosen length.

Partition the sequence into blocks (hands) of length m. For any integer m there are 2^m different possibilities in this block. Count the frequency of each type, i.e. $f_0, f_1, \ldots, f_{2^m-1}$

$$\chi^2 = \frac{2^m}{F}\sum_{i=0}^{2^m-1}(f_i)^2 - F \quad \text{where } F = \max\sum_{i=0}^{2^m-1} f_i \leq \left[\frac{N}{m}\right]$$

The degree of freedom, $\nu = 2^m - 1$

The test can be applied for different values of m and can be based on the word size, e.g. if using 5-bit Baudot code, let m = 5.

2.3.2.4 Autocorrelation Test

The autocorrelation tests for periodicity in a block by comparing the sequence with a shift of itself. It is a measure of how dependent bits of sequence are on each other.

A delay τ is used to detect a possible correlation between bits at a distance τ. Consider a binary sequence s_n of period p. For any fixed τ, the first p terms of s_n and its translate $s_{n+\tau}$ are compared to test how many positions are the same. The autocorrelation function is (simplifying the formula given in [Beker & Piper, p.173]:

$$C(\tau) = \frac{2(number\ of\ coincident\ bits)}{p} - 1 \quad for\ 0 < \tau < p$$

When C($\tau \neq 0$), out-of-phase correlation occurs which will satisfy Golomb's postulate R3.

A sequence, $a_1,\ldots,a_n$ is tested by the following [Beker & Piper, p.173]:

$$A(d) = \sum_{i=1}^{n} a_i a_{i+d} \quad 0 \le d \le n-1$$

The expected value of A(d) is $\mu = \frac{n_1^2(n-d)}{n^2}$

This a computationally intensive test and is a good test to ensure that the sequence is not too *random*.

2.3.2.5 Runs Test

A run is a consecutive string of identical sequence elements. A run of zeros (r_0) is known as a *gap* and a run of ones (r_1) is known as a *block*. This test is only applied after the serial test has been passed.

$$blocks = \sum_{i=1}^{N} r_{1i} \text{ and } gaps = \sum_{i=1}^{N} r_{0i} \text{ where } r_i \text{ are blocks or gaps of length } i$$

From the serial test it can be seen, $n_{01} = r_0,\ n_{10} = r_1,\ n_{00} = n_0 - r_0 \text{ and } n_{11} = n_1 - r_1$

Golomb's postulate R2 can be tested, i.e. half the runs (blocks or gaps) will have length one, quarter of the runs will have length two, etc. A chi-squared statistic can be obtained in the following manner:

$$\chi^2 = \sum_{i=1}^{k} \frac{(r_{1i} - 2^{-i} r_1)^2}{2^{-i} r_1} \quad and\ v = k-1$$

Furthermore, the number of runs is normally distributed with:

$$\mu = 1 + \frac{2 n_0 n_1}{N} \quad and \quad \sigma^2 = \frac{(\mu - 1)(\mu - 2)}{N - 1}$$

The standardised normal score

$$z = \frac{X - \mu}{\sigma} \quad where \ X = runs \ \ and \ \sigma = \sqrt{\sigma^2}$$

The z scores can be looked up in a table of a normal distribution N(μ, σ^2)

2.3.2.6 Binary Derivative Test

A binary derivative is the exclusive-or of adjacent bit pairs in a string [Gustafson, Dawson & Caelli, p.123], [Carroll & Robbins, p.255]. This means that if the adjacent bits are the same (00 or 11) they are replaced by zero and if they are different (01 or 10) they are replaced by a one. Each successive binary derivative drops one bit. For example, consider the sequence:

1 0 0 0 1 0 1 1 1 0 the first derivative will be:

1 0 0 1 1 1 0 0 1

The test function divides the number of ones by the total number of bits. A ratio from $p(0)$ to $p(n)$ is established. The results follow the pattern as given by [Carroll & Robbins, p.257]:

Attribute	Patterned Function	"Random" Function
$p(0) = \frac{n_1}{N}$	variable	close to 0.5
$average = \sum_{i=0}^{n} \frac{p(i)}{(n+1)}$	low	close to 0.5
$range = p(\max) - p(\min)$	high	low

The binary derivative test is not part of the usual statistical tests.

2.3.2.7 Universal Entropy related Statistical Test

Maurer's universal bit test is discussed in a separate section because it is both a recent innovation, and can be used independently of preceding tests, it. See Maurer's universal bit test on page 16.

2.4 Universal Source Coding

This section discusses the idea of universal source coding and leads to the idea of Maurer's universal bit test. If a source can be compressed, it cannot be considered random.

2.4.1 LZ Source Coding

The Ziv-Lempel (LZ) algorithm is an example of a universal source code. The term *universal* in the sense that the code can be used without the knowledge of the source distribution. The algorithm has an asymptotic rate that approaches the entropy of the source. The LZ universal compression algorithm has become a standard for file compression on computers.

A binary string is parsed into the shortest distinct phrases so that there are no identical phrases. For example, the string 1011010100010 is parsed into seven distinct phrases 1,0,11,01,010,00,10.

It can be shown [Cover & Thomas, p.320] that the number of phrases $c(n)$ in a distinct parsing of a binary sequence of length n satisfies:

$$c(n) \leq \frac{n}{(1-\varepsilon_n)\log_2 n} \quad \textit{where } \varepsilon \rightarrow 0 \textit{ as } n \rightarrow \infty$$

$c(n)$ can be regarded as the threshold of complexity. From the Kolmogorov complexity issues, a string is incompressible if it is random. This occurs when $c(n) > \dfrac{n}{\log_2 n}$

It is difficult to compute $c(n)$ especially when n is large, therefore the idea of Maurer's universal test. The LZ compression algorithms do compute $c(n)$, but with the following restrictions: usually the bit size is initially 8 bits and the number of distinct phrases is limited (by the dictionary size). This is not the case when the complexity must be calculated, thus making it difficult to compute.

2.4.2 Maurer's Universal Bit Test

According to Maurer [Maurer, p.90], this test can detect any one of the general class of statistical defects that can be modelled by an ergodic stationary source with finite memory. It is not tailored to detect a specific statistical defect. It is also measures the amount of security by which a cipher system would be reduced when this sequence is used as a key source.

2.4.2.1 Definition of Maurer's Universal Bit Test

A statistic is computed based on the distance between each occurrence of the same *L-bit* code value for each n-bit sequence. The statistic is normally distributed with the expectation and variation being functions of the *L-bit* code size. This test requires large amounts of data, especially as the *L-bit* size increases.

The test quantity f_T is closely related to the per bit entropy H_s of the source when it is modelled by a binary ergodic[7] stationary source with finite memory $M \leq L$. The total length of the sample sequence s^N is:

$N = (Q+K)L$, where

L is the length of the non overlapping blocks into which the sequence is divided,

K is the number of steps of the test,

Q is the number of initialisation steps.

The test function is defined by $f_{T_U}(s^N) = \frac{1}{K} \sum_{n=Q}^{Q+K-1} \log_2 A_n(s^N)$

where for $Q \leq n \leq Q+K-1$, $A_n(s^N)$ is defined by an integer valued function

$$A_n(b_n(S^N)) = \begin{cases} n & \textit{if there exists no positive } i \leq n \textit{ such that } b_N(s^N) = b_{n-1}(s^N), \\ \min\{i : i \geq 1, b_n(s^N) = b_{n-i}(s^N)\} & \textit{otherwise} \end{cases}$$

$b_n(s^N) = [s_{Ln}, \ldots, s_{Ln+L-1}]$ denotes the nth block of length L. The sequence is scanned for the most recent occurrence of block $b_n(s^N)$.

A distribution of a random sequence is used to specify the acceptance or rejection regions. The mean and variance of a single term $\log_2 A_n(R^N)$ of $f_{T_u}(R^N)$ can be computed as $Q \to \infty$ [Maurer]. The expected value of the random sequence is the same as the expected value of the test sequence, i.e. $E[f_{T_U}(R^N)] = E[\log_2 A_n(R^N)]$.

[7] A process is stationary when all the probability densities of a random process are time independent. An ergodic process has the property that a random process observed for a fixed time is identical to observing one sample function all the time. A process cannot be ergodic unless it is stationary. Whereas, a stationary process is not necessarily ergodic.

The resulting threshold is then $t = E[f_{T_U}(R^N)] \pm y\sigma$. Maurer gives all the values for l-bit blocks in the range 8 to 16.

When the per-bit entropy is close to one, the test requires a much longer sample sequence than the frequency test or other similar test. Increasing the length of the sample sequences reduces the standard deviation and therefore allows detection of smaller deviations from a binary symmetric source. Another reason that the LZ source coding algorithm is not used, is that is difficult to define a test function so that $f_{T_u}(R^N)$ can be computed [Maurer, p.101].

2.4.2.2 The "Universality" of the Test

There is a known failure of this test as it will pass the RANDU[8] algorithm. If L can be such that $L = 31$, then the test will fail RANDU and other similar generators. With $L = 31$, an array of 2^{31} elements is required and the algorithm must also test so may iterations. This would take a very long time on current computer technology.

2.5 Conclusion

There are many methods to check the randomness properties. This research has mentioned a few of the more well known methods. There is much ongoing research in the field, e.g. Yao's next bit test, spectrum analysis and avalanche testing.

A suite of tests has been developed based on the methods described to detect non randomness [Stevens 1994].

The question of why "randomness" is so hard to reproduce or measure can be summed by the following quotation of John Taber on the newsgroup sci.crypt:

> *Attempting to observe randomness in any way distorts random events. Randomness exists so long as you can't see it. When you see it, it is no longer random.*

[8] RANDU is a linear congruential RNG of the form $X_{n+1} = (65539 * X_n) \bmod 2^{31}$, [Knuth]. RANDU is known to be a particularly poor generator.

Chapter 3

DATA COMPRESSION

3.1 Scope

This chapter introduces the concepts of lossless data compression. It is not intended as an in depth discussion, but merely to lay the ground work for later chapters.

3.2 Introduction

Data can be compressed whenever some events (or symbols) are more likely than others. The basic idea of compression is to assign shorter codewords to more probable events and longer codewords to less probable events. Compression removes some of the redundancy from a source or message.

3.2.1 Types of Data Compression

There are two forms of data compression, namely: lossless and lossy.

3.2.1.1 Lossless

No information is lost in the compression process. The original source data is reproduced exactly by decompressing the compressed stream. In this report, only lossless compression is considered and thus all references to compression refer to lossless compression.

3.2.1.2 Lossy

Information is discarded in the compression process and thus the original data cannot be recovered by the decompression process. Instead, decompression produces an approximation to the original data. The compression ratio is then dependent on the degree of approximation. Generally, this type of compression is used for compressing wave form data, e.g. audio and video.

Some forms of lossy compression rely on the perceptive techniques of the human senses. An example of this is the standard telephony codecs to fit a speech signal into a limited bandwidth and the speech compression method used in cellular telephones.

3.2.2 Entropy

In the field of Information Theory, entropy[9] represents the intrinsic information content of the source. In this case predictable events convey no information, whereas unpredictable events convey a large amount of information. Entropy is thus the measurement of the information content in a message. The higher the entropy, the greater the information content of the message.

Entropy is the lower bound for compression [Bell, Cleary & Witten, p.47]. With lossless compression, the source cannot be compressed below its entropy because it must have at least that amount of information content. This is known as the *Noiseless Source Coding Theorem* which was proved by Shannon [Shannon: Mathematical Theory of Communication, p27].

Shannon defined entropy as the smallest number of bits required to encode an event. The entropy of a discrete random variable *X* is defined by [Cover & Thomas, p.13]:

$$H(X) = -\sum p(x)\log_2 p(x)$$

The entropy can also be considered in terms of uncertainty [Lucky]. Low predictability (thus high uncertainty) results in a high entropy.

3.3 Data Compression = Modelling + Coding

The data compression function can be broken into two separate entities[10], namely the model and the coder [Rissanen & Langdon]. This insight was considered a major advance in the field of data compression.

- *Modelling* is the process of representing the source to be compressed by assigning probabilities to the source symbols.
- *Coding* is the process of mapping the model's representation of the source into a compressed bitstream.

9 In physics, entropy is a measure of the amount of disorder or uncertainty in a system [Irvine].

10 This heading "Data Compression = Modelling + Coding" is taken from [Nelson] as it clearly shows the components of the data compression function.

3.3.1 Modelling

Modelling is the key to effective data compression. In general, entropy is based solely on the model. The amount of compression (or how well the source can be compressed) is directly related to how well the model represents the entropy of the source. Incorrect statistics can lead to a code string with more bits than the original.

The following diagram indicates the relationship between the model and coder. Both the encoder and decoder models must follow one another, otherwise synchronisation will be lost between the models.

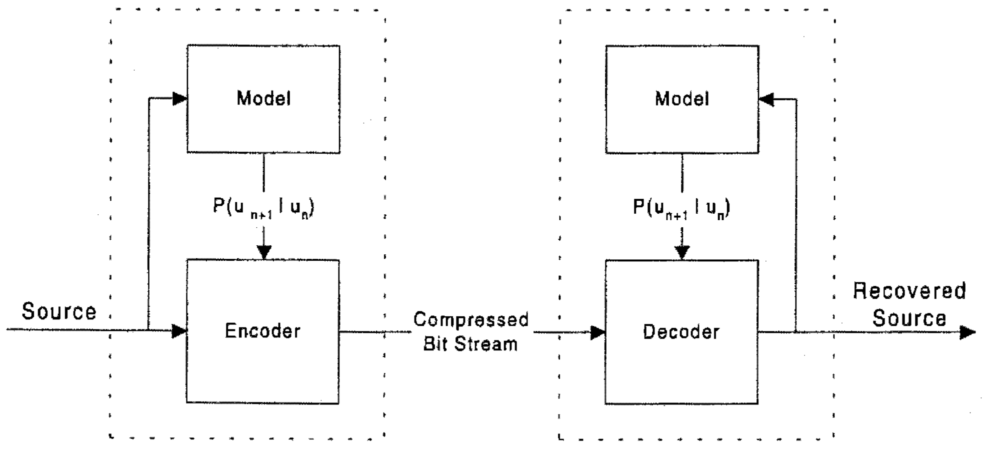


Figure 1. Relationship between the model and coding sections

A model is an estimator which provides the encoder with the conditional probabilities for the next symbol. The decoder model makes the same estimation based on the symbol which has been decoded.

A simple model has a fixed probability for a symbol irrespective of its position in the message. This type of model is referred as an *order-0* model. Models may have different contexts associated with each symbol. In an *order-1* model, the context of the current symbol depends on the immediate previous symbol, e.g. in English, it is highly probable that a '*u*' will follow a '*q*'.

Models may be adaptive, semi-adaptive or non adaptive. In the adaptive model, the symbol probabilities will change dynamically as the source text is processed. A semi-adaptive model may make a preliminary pass of the source to gather statistics. This model data must then be transferred to the decompressor. A non-adaptive model uses fixed probabilities that are hard coded into the compressor and decompressor. Any deviations of the source from this predefined model will cause a loss in compression.

The type of compression technique will influence the complexity of the model. These techniques will be discussed in the following section. For example:

- Updating an adaptive Huffman coder is processor intensive [Nelson, p.83],
- Dictionary based methods have the model closely intertwined with the coder, and
- The arithmetic coder has a clean separation between the model and coder.

Modelling will be discussed in more detail in Chapter 5.

3.3.2 Coding

The coder accepts the events to be encoded. A prime feature of any coder is that it should have the first-in-first-out (FIFO) property [Langdon, p.136]. This means that events can be decoded in the same order as they were encoded.

Another desirable feature is that of incremental coding [Bell, Cleary & Witten, p.114]. The coder can output codewords before it has processed the entire input stream. A codeword can be output as soon there are no bits that might change i.e. it no longer has any influence on the calculations. A similar process also applies to the decode process.

The next section on Compression Techniques will show some of the many forms of coding.

3.4 Compression Techniques

There are several compression techniques that fall under three classes. These include the intuitive *ad hoc* methods, the statistical methods and the dictionary (also known as textual substitution) methods [Bell, Cleary & Witten, p.206].

3.4.1 Ad Hoc Coding Methods

These coding methods do not fall into one of the two main methods of compression, namely statistical and dictionary methods. Some of these ad hoc methods include run length, move to the front, differential, bit mapping, packing, special data types and even irreversible compression (e.g. removable of white spaces in text) [Bell, Cleary & Witten, p.20]. A few of these methods will now be described.

3.4.1.1 Run Length Encoding (RLE)

This is a simple method where runs of identical symbols are encoded as a symbol and a count. It is often a precursor to additional compression methods.

3.4.1.2 Move To Front (MTF)

The appearance of a word in the input makes it more likely to occur in the near future. This is known as *locality of reference*. Each word is put on a stack, with words near the top assigned shorter codes. If the word is already on the stack, its position (code) is output and it is moved to the top of the stack.

3.4.1.3 Burrows Wheeler Transform

This method is a relatively recent development [Fenwick, 1996]. The Burrows Wheeler Transform (BWT) transforms a block of data into a format that is well suited to compression. The text is processed in blocks and reordered. The resulting output block contains the same bytes as in the input block but with a different byte ordering. This ordering is reversible. The text is then compressed with standard compressors. Usually the first compression step is to apply run-length encoding or move-to-the-front and follow this by some standard entropy encoder such a Huffman source coder.

The BWT uses the following context in contrast with other adaptive methods which use the preceding context. This means that the transform can only operate on blocks that are resident in memory in contrast to other methods which can operate on a stream.

The compression performance is comparable with the best higher order statistical compressors.

3.4.2 Statistical Coding Methods

These are compression methods where the code is determined by the estimated probability of the input symbol. Given these probabilities, a code table can be generated with some important properties:

- Different codes have different numbers of bits.
- Low probability codes have more bits than high probability codes.

- As codes have the *prefix free*[11] property [Cover & Thomas, p.81], they can be uniquely decoded although they have different numbers of bits. No codeword is a prefix of another code word, i.e. the decoder cannot confuse two codewords as each codeword is uniquely decodable.

These methods can be considered as the coder section of the whole compression process. The model has assigned the necessary symbol probabilities. A few statistical methods will be described in the following sections.

3.4.2.1 Shannon-Fano Coding

Messages are coded according to their probabilities. This algorithm builds the tree from the root [Bell, Cleary & Witten, p.103].

1. In decreasing probability order, list all possible messages.
2. Divide the list into two sections with equal probabilities.
3. Start the code for the first section with a '0' bit and those in the second section with a '1' bit.
4. Recursively apply the above procedure until each section contains a single message.

The average code length lies between [H, H + 1) where H is the entropy[12] [Bell, Cleary & Witten, p.104].

3.4.2.2 Huffman Coding

Huffman codes are similar to Shannon-Fano codes. A Huffman tree is built from the leaves in contrast to the Shannon-Fano which builds the tree from the root [Bell, Cleary & Witten, p.105].

1. List all possible messages with their probabilities.
2. Locate the two messages with the smallest probabilities.
3. Replace these by a single set containing both, whose probability is the sum of both.
4. Repeat until the list contains one member.

[11] A prefix code is a code which satisfies the Kraft inequality. The Kraft inequality provides the proof that a code is uniquely decodable if the code lengths are greater or equal to that determined by the entropy. For a D-ary alphabet the lengths l_i of the codewords must satisfy $\sum_i D^{-l_i} \le 1$ [Cover & Thomas, p.82].

[12] The notation $[a, b)$ denotes a half open interval such that $a \le x < b$.

As each list contains only two members it can be represented by a binary tree with the original message at the leaves. The tree is then traversed from the root to the message outputting a '0' for left branch and a '1' for a right branch.

An example of Huffman coding (based on a fixed model) is as follows:

Consider a message of *ABABACA*, that is four *As*, two *Bs* and one *C*.

4	2	1
I	I	I
A	B	C

Combine the least probable symbols into one, i.e. *B* and *C* with a new count of 3.

4	3
I	I
A	B + C

Combine the least probable symbols into one, i.e. *A* and *B*+*C* with a new count of 7.

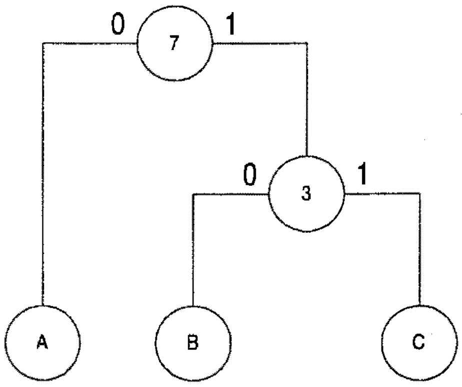


Figure 2. Example of a Huffman Tree

Starting at the root of the tree, encode a zero bit for left traversal and a one bit for right traversal. So *A* is encoded as a 0 and a *B* is encoded as a 10. The entire message is encoded by a bit stream of length ten bits, 0100100110. A binary code would generate a message of fourteen[13] bits long.

[13] Message length of seven characters each requiring two bits to uniquely represent each character.

The code generated is prefix free and can be uniquely decoded. It is also instantaneous as decoding can take place without look ahead, that is a symbol can be recognised as soon as it is complete.

Like Shannon-Fano Coding, Huffman Coding can take up to one extra bit per symbol (unless probabilities are exact powers of ½). The average code length is bounded by [H, p + 0.086) where p is the probability of the most likely symbol and H is the entropy [Bell, Cleary & Witten, p.107].

Huffman coding is usually done over blocks of bits. This *blocking* [Bell, Cleary & Witten, p.107] allows the entropy to be approached when the symbol probabilities are not binary powers. In practice, ideal blocking is difficult to achieve in real-time. Large blocks are required to get optimal compression. These problems can be solved with arithmetic coding.

3.4.2.3 Arithmetic Coding based on Floating Point Arithmetic

Arithmetic coding[14] forms the basis of this research and as such will be presented in more detail than any of the other compression coding methods. Chapter 3 will consider the subject in more detail. This section will introduce the concept of arithmetic coding.

Arithmetic coding falls into the class of statistical coding methods. It is a method of writing a code in a non-integer length allowing the code to approach the ideal entropy. Arithmetic coding is superior in most respects to Huffman coding for the following reasons:

- It is a true entropy coder as it achieves the theoretical entropy bound for any source [Bell, Cleary & Witten, p.108] and [Howard & Vitter 1992, p.86]. No blocking of input data is required so that the entropy of the source can be attained *(assuming that the model is entirely representative of the source).*
- The probabilities are not required to be arranged in any order (as in Huffman Coding or Shannon-Fano) [Bell, Cleary & Witten, p.108].
- Codes do not have to be an integer number of bits long [Nelson, p.123].
- There is a clear separation between the model representing the data and the coder. The complexity of the model has no affect on the coder.
- It lends itself towards adaptive models *(because of the above point)* [Bell, Cleary & Witten, p.121].
- Code symbols can be continuously generated for infinitely long source sequences.

[14] It is interesting to note that virtually all forms of arithmetic coding are protected by patents.

However, arithmetic coding does have some disadvantages.

- Arithmetic coding tends to be slow due to the arithmetic[15] operations and the need to maintain a cumulative[16] probability table.
- It does not produce prefix free codes and thus does not allow parallel processing [Howard & Vitter, p.86].
- It requires an end of stream indicator [Howard & Vitter, p.86].
- There is poor error resistance[17] as the code is not a prefix free code. This fact is prevalent in most compression methods[18] as well.

Codewords representing source symbols can be viewed as *code points* in the half open unit interval [0,1). These code words divide the unit interval into non-overlapping sub intervals. Each code word is equal to the *cumulative sum* of the preceding symbols. As each code word is transmitted, the range is narrowed to the portion of it allocated by the symbol. The reduction of the interval is based on the probability of the symbol generated by the model. More likely symbols will result in a smaller reduction of the interval than less likely symbols.

Incremental coding is performed without enumerating all the messages in advance. This means that as soon as the number of bits that uniquely represent the code point are generated, they can be sent to the decoder.

The basic algorithm (given by [Howard & Vitter 1992, p.87]) is as follows:

1. Begin with the current interval initialised to [0, 1)
2. For each event in the file perform these two steps:
    - Subdivide the current interval into a subinterval, one for each possible event,

[15] Some of the arithmetic multiply and divide operations can be replaced with shift and add operations with some compression loss. This concept will be explained in the following chapter.

[16] A recent innovation speeds up this table manipulation with a new data structure [Fenwick 1994]. Implementors refer to it as the Fenwick tree [Bloom].

[17] Arithmetically coded bit streams can be protected against channel errors by adding in controlled redundancy [Boyd, Cleary, Irvine, Rinsma-Melchert & Witten].

[18] In contrast, a fixed model Huffman coder can recover after an error. As soon as the next prefix code is found synchronisation will be attained with only the loss of some input symbols.

- Select the subinterval corresponding to the event that occurs next and make this the new current interval.

3. Output event bits to distinguish the final interval from all other possible final intervals. This does lead to a small message termination overhead that is insignificant [Bell, Cleary & Witten, p.121].

Longer messages result in a smaller interval and thus require more bits to represent this smaller interval than a large interval. The encoding example in the next section shows how the interval is split into smaller and smaller intervals.

3.4.2.3.1 Arithmetic Encode

The pseudo code for this encode operation is as follows [Nelson, p.126]:

```
$range = high - low$
$high = low + range \times cumulative\ probability[upper]$
$low = low + range \times cumulative\ probability[lower]$
```

where *upper* is the upper limit of the symbol's probability, and *lower* is lower limit. These three steps are collectively known as the *coding step.*

Using the same probabilities as in the Huffman example and the same message, *ABABACA*, an arithmetic coding example based on a fixed model using the above algorithm is as follows:

Character	**Probability**	**Interval - cumulative probability**
A	$\frac{4}{7} = 0.5714$	[0, 0.5714)
B	$\frac{2}{7} = 0.2857$	[0.5714, 0.8571)
C	$\frac{1}{7} = 0.1428$	[0.8571, 1)

Each time the algorithm is applied, the following steps take place for the message.

Character	New Interval [low, high)	Range
A	[0, 0.5714)	1.0
B	[0.3265, 0.4897)	0.5714
A	[0.3265, 0.4197)	0.1632
B	[0.3797, 0.4064)	0.0932
A	[0.3797, 0.3950)	0.0267
C	[0.3928, 0.3950)	0.0153
A	[0.3928, 0.3941)	0.0022

Any value in the interval [0.3928, 0.3941) will represent the entire message. The message can be represented by the binary fraction $(0110010011)_2$ which is equivalent[19] to the decimal fraction value of $(0.3935)_{10}$. As can be seen, each successive character will narrow down the interval. In this particular example, ten bits are used to represent the message as in the Huffman example.

The following figure shows part of the interval shrinking process for the message portion *ABABAC*.

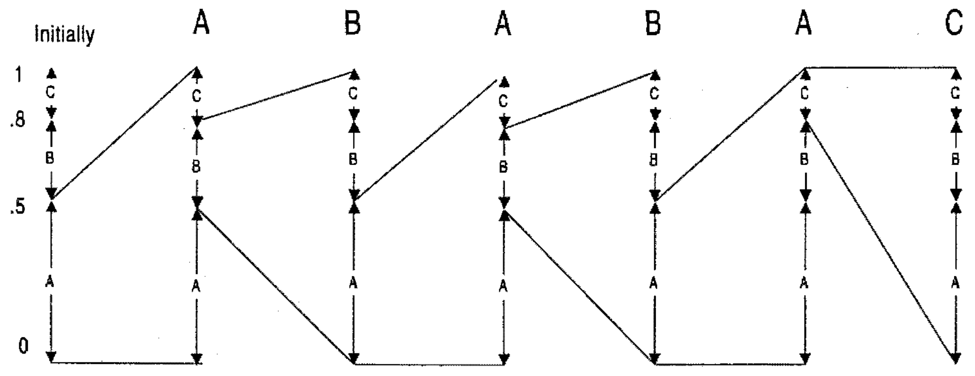


Figure 3. Shrinking the arithmetic coding interval

In the case of adaptive arithmetic coding with the above example, the probabilities of a character would be different the next time the same character is coded.

[19] The value of 0.3935 is determined by summing the binary fractions until a value within the final interval is found. For example: 0110010011 is represented by $\frac{1}{4}+\frac{1}{8}+\frac{1}{64}+\frac{1}{512}+\frac{1}{1024}$

3.4.2.3.2 Arithmetic Decode

Decoding applies the reverse operation. Find the first symbol in the message by investigating what symbol owns the space within the interval. From the example, 0.3935 falls in the interval [0, 0.5714), so A is the first character. Since the range of A is known, A can be removed. The interval will now encompass B and thus the entire process can be reversed.

The reverse decoding step algorithm [Nelson, p.127] is:

$$\begin{aligned}
&symbol = symbol\ which\ is\ straddling\ the\ input\ value \\
&range \ = cumulative\ probability[upper] - cumulative\ probability[lower] \\
&value \ = (value - cumulative\ probability[lower]) / range
\end{aligned}$$

Although the above examples are explained in terms of floating point arithmetic, standard integer arithmetic is used in practice. There are a number of different forms of the arithmetic coder. Some of these methods are given by [Ekstrand]. The next chapter will describe a shift and add arithmetic coder. This implementation requires neither multiplication nor division.

3.4.3 Dictionary Methods

The main concept of the dictionary method is that phrases (substrings) of the text are replaced by tokens [Bell, Cleary & Witten, p.207]. The value of the token is assigned from a dictionary of tokens. Although there are static, semi-adaptive and adaptive dictionary coders, we will only consider the adaptive versions. Two key papers by Ziv & Lempel were the cornerstones of the dictionary methods. The LZ77 method is adopted from their paper in 1977 and the LZ78 methods from their paper in 1978 [Bell, Cleary & Witten, p.217]

In general, these methods replace phrases with a pointer to where they have previously occurred in the text. Ziv-Lempel methods adapt easily to new text and novel words and phrases are constructed from parts of earlier words. The LZ methods achieve good compression and are fast in comparison to the standard statistical coders. The LZ methods are a class of *asymptotically optimal*[20] codes.

There are many variants of the LZ methods, the list of which is continually growing. Most differences arise in the parsing strategy, and the method of constructing and refreshing the dictionary. Although the

[20] A code is *asymptotically optimal* when $H \to 1$ as the length approaches infinity for a given probability distribution.

Ziv-Lempel[21] concept seems vastly different from the statistical coding methods, there is an algorithm to convert any dictionary coder to that of a statistical coder [Bell, Cleary & Witten, p.248].

Most commercial compression packages are based on one of the LZ variants. Often dictionary methods are referred to as textual substitution, codebook coding or macro coding schemes.

3.4.3.1 LZ77 Based (Sliding Window)

This was the first form of LZ compression. The main data structure is a text window divided into two parts. The first part consists of a large block of recently decoded text. The second (normally much smaller) is the look ahead buffer that contains input symbols not yet encoded. The algorithm tries to match the contents of the look ahead buffer to a phrase in the dictionary. The longest match is coded into a triple containing: an offset to the phrase in the text window, the length of the phrase and the first symbol in the look-ahead buffer that follows the phrase.

LZ77 has a performance bottleneck as it has to do string comparisons against the look ahead buffer for every position in the text window. Increasing the size of the dictionary further decreases performance. Decoding is faster than encoding. [Bell, Cleary & Witten, p.219], [Nelson 1992].

3.4.3.2 LZ78 Based

The input source is parsed into phrases where each phrase is the longest matching phrase encountered previously plus one character. Each phrase is encoded as an index to its prefix plus the additional character. The new phrase is added to the list of phrases. As there is no text window, there is no restriction on how far back the pointer can reach.

LZW - This adaptation was described by Welch [Welch 1984]. Only the pointer is sent as the explicit character is eliminated. This method is popular as it is simple to implement and is not processor intensive.

The LZW algorithm begins by initialising the dictionary with every character in the input alphabet. Thus, with an 8 bit character, there will be 256 characters in the alphabet and the first 256 entries in the dictionary will correspond to these characters. LZW always outputs tokens for phrases (strings) that are already in the dictionary. The last character (symbol) of each new phrase is encoded as the first character of the next phrase.

[21] Authors tend to refer to Ziv-Lempel when writing their names in full and to LZ when describing a particular method.

A simple example explains this concept. Consider the sentence *THIS IS*

	Input	Dictionary	Output
1	*T*	*T* is in dictionary as initial token	
2	*H*	Search for '*TH*'. Not found, add 256 = '*TH*'	T token
3	*I*	Search for '*HI*', not found, add 257 = '*HI*'	H token
4	*S*	Search for '*IS*', not found, add 258 = '*IS*'	I token
5	*sp*	Search for '*Ssp*', not found, add 259 = '*Ssp*'	S token
6	*I*	Search for '*spI*', not found, add 260 = '*sp*I'	sp token
7	*S*	Search '*IS*', found	
8	*sp*	Search '*ISsp*', not found, add 261 = '*ISsp*'	token 258

Figure 4. Example of LZW compression

The token bit size depends on the dictionary size. LZW uses a fixed token size of 12 bits that results in a dictionary of 4096 entries.

Other variations of LZ78 include:

- LZC, LZT, LZMW, LZJ, LZFG and hosts of others. V.42 bis (modem compression standard) uses a mixture of most of these methods including LZW.
- LZC - UNIX *compress* is based on this method. The tokens represent pointers of various bit lengths.
- LZT - Once the dictionary is full, LRU replacement is employed to recover dictionary entries.
- LZJ - The string length is limited which means that the trie[22] is depth limited.

3.5 Conclusion

This chapter has considered a number of different forms of compression. Both statistical and dictionary methods were shown. Arithmetic coding will be discussed in more detail in the following chapter. The concept of arithmetic coding will be used for the coder section in the implementation of the universal homophonic coder.

[22] A trie is a multiway digital search tree with a path from a root to a node for *each* string in the tree.

Chapter 4

ARITHMETIC CODING

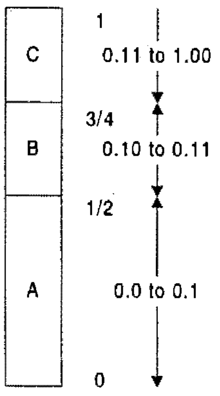
4.1 Introduction

The previous chapter considered various forms of lossless compression techniques. This chapter will consider arithmetic coders in more detail. The concept of arithmetic coding was introduced with floating point values. This chapter will show how the implementation can be achieved with integer arithmetic.

4.2 Binary Arithmetic Coding

In the floating point description of arithmetic coding, the interval [0.00, 1.00) was considered. This can be written as [0.00, 0.99...) since it has been shown[23] (by many mathematicians) that $1 \equiv 0.9999999.........$ The same fact applies when the interval is considered as a binary range [0.00, 0.111111).

Consider three symbols *A*, *B*, *C* from a three symbol alphabet with probabilities ½, ¼, and ¼ respectively. The binary representation of ½ is 0.1 and that of ¼ is 0.01. This is shown in the initial interval diagram as



[23] $0.999... = \sum_{n=1}^{\infty} \frac{9}{10^n} = \lim_{n\to\infty} \sum_{n=1}^{m} \frac{9}{10^n}$

Any interval in the binary range 0.0 to 0.1 can specify the symbol A. For example, the first quarter is still in the range of symbol A and can be represented by binary 0.01. However, this would mean outputting extra bits. The shortest number of bits is always sent and in this case 0.0 would be sent. (*The splitting of the interval was perceived as an important concept for homophonic coding, e.g. the symbol A, can be represented by any code word as long as it is in the range of 0 to ½*).

It should be noted that the symbol ranges and the coding interval are *not* the same. They are both different 1-dimensional spaces. Any part of the range may be used to specify the current symbol. However, any part of the range cannot be used in the coding step to calculate the new range. This means that the decoder must know about the range for a symbol.

4.2.1 Practical arithmetic coding

To ensure fast compression or decompression using fixed point arithmetic, the precision required to represent the [*low, high*) interval increases with the length of the message. The interval should be kept within the word size of the computer. As the number of input symbols increases, the interval decreases and becomes too small to fit in the finite word size of the machine.

We show this with the following example. Consider a machine with a word size of 32 bits, the precision is limited to 2^{-32}. With each byte being represented by eight bits and having an equal probability of $\frac{1}{256}$, encoding the first event will result in an interval of $\frac{1}{256}$ or 2^{-8}. The second byte will reduce the interval to 2^{-16}. After three bytes the interval is 2^{-24} and after four bytes the interval is 2^{-32}. The fifth byte will cause an overflow.

Incremental operation will help to overcome this precision problem. The most significant bits can be output whenever they are no longer susceptible to further change. This normalisation operation of scaling the interval when the bits are output raises potential underflow and overflow problems.

These issues and how to signify the end of the input stream will have to be solved as discussed below.

4.2.1.1 Scaling

When the code range narrows, the most significant bits of the interval *high* and *low* will become the same. Bits are output when any of the upper bits of both low and high are the same. The interval can now be scaled by a left shift (multiply by two).

4.2.1.2 Underflow

The scaling operation can cause the *low* and *high* to become very close to each other. For example: the interval may become [0.011111, 0.100001). No matter how the interval is scaled, the most significant bits will never become the same (in a finite word length register) and the encoding operation would not be able to continue.

The encoder must guarantee that the interval is large enough to prevent this underflow problem. Bell, Cleary & Witten proposed a unique method to overcome this problem [Bell, Cleary & Witten, p.116]. They constrain the maximum allowed cumulative frequency count so that is always less than the interval.

4.2.1.3 Overflow

Integer multiplication can cause an overflow in the coding step with the range as one of the multiplicands. The maximum allowed symbol frequency multiplied by the range must fit into the integer word of the machine [Bell, Cleary & Witten, p.118].

4.2.1.4 Word Length Constraints

The constraints imposed on the finite machine word by the underflow and overflow can be represented by the following equations [Bell, Cleary & Witten, p.118]. The frequency counts are represented by *f*, the machine word length by *c*, and the precision by *p*:

$$f \leq c - 2$$
$$f + c \leq p$$

If p is restricted to a precision of 16 bits, then $c = 9$ and $f = 7$ making it impossible to encode a 256 symbol alphabet.

Generally, the word length is sixteen bits, so $c = 16$. In this case the maximum frequency count will be limited to fourteen bits, $f = 14$. Thus for the precision, $p \geq 30$, the computer must support 32-bit signed integers.

4.2.1.5 End of the stream

The decoder needs to know when to stop decoding. This can be done in one of three ways:

1. Passing the length of the message to the decoder before decompression takes place. This method is not well suited to adaptive methods as the whole input stream must be parsed before compression can take place.

2. Ensuring that the final decode arithmetic operations on the coding step terminate in a null.

3. The compressor sends an EOF (end of file) symbol to the decompressor at the end of the message. This is the most effective method in an adaptive system.

4.2.1.6 Compression Efficiency

In theory, arithmetic coding should be virtually perfect and match the entropy of a given model exactly. There are some factors which can cause a reduction in the compression performance.

- *Message termination overhead.* After the final symbol is encoded, additional bits must be output to disambiguate the final symbol. In practice, the stream is usually blocked into 8-bits and will have to be flushed with additional bits when the output is not a multiple of this size. This may then cause an overhead of 9-bits [Bell, Cleary & Witten, p.129].

- *Use of fixed length arithmetic.* The scaling of the counts so that $f \leq c-2$ is satisfied. Bell, Cleary & Witten [Bell, Cleary & Witten, p.129] showed that the loss of compression was negligible and in the order of 10^{-14} bits per symbol.

4.2.2 Fast arithmetic coding without multiplication or divide operations

Langdon [Langdon, p.137] described a form of arithmetic coding based on *shift* and *add* operations for the encode process and *shift* and *subtract* operation for the decode process. Any integer is the sum of the powers of two. It is easy to apply this fact to only use shift operations.

As in the previous method, codewords (representing source symbols) can be viewed as *code points* in the half open unit interval [0, 1). These code points divide the unit interval into sub intervals, with the following properties: [Langdon, p.138]

- Each code point is the sum of the probabilities of the preceding symbols (cumulative probabilities). Each code point will be referred to by the letter *C*.

- The width or size of each subinterval corresponds to the probability of the symbol. This interval will be referred to by the letter *A* (other authors use the symbol *R* to denote the range).

- By restricting the interval widths to negative integer powers of two, multiplication is replaced by a shift to the right. These negative integer powers of two probabilities will be referred to by the term *dyadic probabilities*. This restriction will cause a loss in compression of less than four percent [Williams, p.56].

4.2.2.1 Fast Arithmetic Encode

A new code point is the sum of the current code point C, and the product of the interval width and cumulative probability $P(v_i)$ of the symbol v_i being encoded. The operation can be written as $C_k = C_{k-1} + A_{k-1} \times \sum P(v_i),\ k = 1,2,3,\ldots\ \textit{with}\ C_0 = 0\ \textit{and}\ A_0 = 1$

The interval width A of the current interval is the product of the probabilities of the symbols already encoded. The new interval width of the symbol v_i is then $A_k = A_{k-1} \times P(v_i),\ k = 1,2,3,\ldots\quad A_0 = 0$

The value $A_{k-1} \times \sum P(v_i)$ is often referred to as the *augend* [Langdon, p.138].

4.2.2.2 Fast Arithmetic Decode

The decode process is the reverse of the encode process and consists of the following steps:

Step 1: Decoder C comparison. The codeword is examined to determine in which interval it lies. This interval represents the symbol.

Step 2: Decoder C adjust. The augend value for the symbol is subtracted from the codeword,

Step 3: Decoder C scaling. C must be rescaled by undoing the multiplication for A.

4.2.2.3 Problems with the shift and add method

Rubin [Rubin, p.673] showed that this shift and add method had a number of problems which would increase the encoding computation time to $O(n^2)$ for a string of length n. The arithmetic operations must be carried out with n digits of accuracy, or multiple precision techniques must be used.

Rubin proposed a new algorithm for this method. Unfortunately, this algorithm makes the implementation non-dyadic and drastically increases the number of arithmetic operations.

Langdon extended the basic shift and add algorithm into the skew coder algorithm, which has been patented and is known as the *Q*-coder.

4.3 Conclusion

This chapter considered arithmetic coding in more detail. Adaptive arithmetic coding relies on the model for accurate prediction of probabilities. The coder unit is totally independent of the model and any form of model may be used; e.g. static, semi-static or adaptive. Modelling will be considered in the next chapter.

Chapter 5

MODELLING

5.1 Introduction

The data compression function comprises two separate functions: modelling and coding. The coder will only code what is presented by the model. To achieve maximum compression of a source, both a good model and a way of representing the model are required. To achieve this a model must accurately predict the probabilities in the input stream [Nelson, p.167]. This prediction must deviate from a uniform[24] distribution.

5.2 Models

A model provides the probability distribution of the next character in any context. The simplest models are insensitive to the next character and give the same distribution regardless of the next character. These models are known as *order-0* models or *memoryless* models [Langdon, p.135].

There is no obvious mathematical relationship between the probabilities of different characters and therefore most models estimate the character probabilities individually. The estimate for the probability is given by estimate of the number of occurrences of the character:

$$p = \frac{\textit{weight of character}}{\textit{total weight of all characters}}$$

Any loss in compression is attributed to the modelling overhead [Boyd 1991, p.276]. Modelling overhead is due to incorrect probabilities being assigned by the model. Perfect compression could be attained with no modelling overhead and an entropy coder such as arithmetic coding. In practice perfect compression is not achievable.

[24] If the model assigned the same probability, say 2^{-8} for each input character of a 256 character alphabet, then there would be no compression as the output will be the same size as the input. Each character will still require eight bits.

The encoder and decoder must maintain synchronisation at all times. There are three methods of achieving this: fixed, semi adaptive or adaptive. Each one of these model may be of different orders, i.e. based on the conditional probabilities of the current symbol. A model that always uses i previous characters to predict the current character is referred to as an *order-i context model* [Hirshberg & Lelewer, p.115]. This model consumes more memory as i increases and its efficiency decreases [Bell, Cleary & Witten, p.155].

Each context consumes a large amount of processor memory. For example, an order-1 model with an alphabet of 256 characters will required 256 contexts for each character, resulting in a memory size of 65535 characters.

An order-(-1) model has all the symbols assigned the same probabilities [Bell, Cleary & Witten, p.34]. A model need not be character based, words and bits can be used as well.

In all discussions of models, the same model will be used in the encoder and decoder.

5.2.1.1 Fixed or Static Model

This model does not change while the stream of symbol is coded and the same model is used for all different types of input. For example, a fixed model for English will have a high probability of the letter *e* occurring. Deviations of the input text based on the model will cause a loss of compression.

5.2.1.2 Semi-adaptive or Hybrid Model

In this form of model, multiple passes of the source are required. The first pass will identify a suitable model. The second pass will use the model for optimum coding. The model produced (or selected) for the input is transmitted to the decoder before the compressed stream is sent. The decoder can now use the same model to restore the source stream.

Long input streams may cause a loss of compression due to the averaging effect of the counting the input symbols.

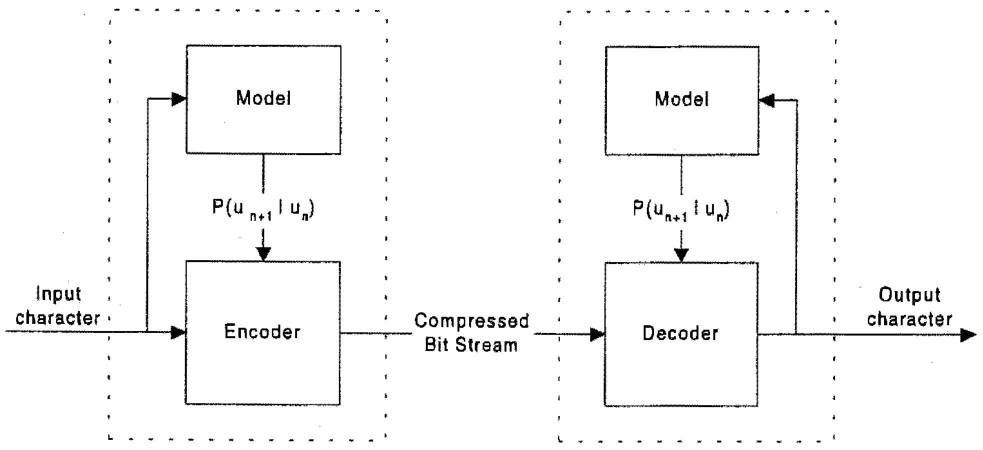
5.2.1.3 Adaptive or Dynamic Model

The adaptive model adapts to the input source during a single pass (i.e. in effect learning the model). The adaptation process is achieved by increasing the probability of each symbol encoded. Both the encoder and decoder models must be initialised to use the same model. As the code is decoded, the decoder alters the model accordingly and in sequence with the new probability assignments of the encoder model.

A simple order-0 model may assign an initial value to all symbols in the alphabet, for example unity. For each new symbol read, the probability for that symbol is incremented. Initially, the model will track the source slowly and for short input sequences the compression will not be as good as longer sequences.

In an adaptive model there are two items that can be altered: the probabilities associated with a particular conditioning class and the set of conditioning classes. [Rissanen & Langdon 1981, p.12].

The block diagram of the adaptive model is repeated here.



Adaptive compression steps are given with the following pseudo code:

```
InitialiseModel()
while(input characters are available)
{
  encode(input character)
  UpdateModel(input character)
}
```

The decompression function is applied in reverse as follows:

```
InitialiseModel()
while(decode symbols available)
{
  OutputCharacter(decode symbol)
  UpdateModel(output character)
}
```

An model must consider the following issues:

5.2.1.3.1 Locality of reference

Strings tend to occur in clusters. What has been seen in the recent past is more likely to be seen again. This is known as locality of reference or recency effect [Howard & Vitter 1992, p.94]. Thus more weight can be assigned to recent occurrences of the symbols.

5.2.1.3.2 Rescaling

In the normal operation of an adaptive model, the count values associated with each context are being incremented and stored as fixed binary integers. Before any of these values overflow, they must be scaled down, e.g., by halving the count [Nelson, p.95] and [Bell, Cleary & Witten, p.151].

Scaling often improves compression and locality of reference ensures that recent statistics are given more weight.

5.2.1.3.3 Zero frequency problem

A symbol will have a zero probability when it not yet been encountered and since symbols are encoded in *-log p* bits, the code length approaches infinity. This means that all counts for a context must be initialise to some value other than zero. In our implementation we initialise the adaptive model so that each symbol has a count of one.

5.2.1.4 Entropy differences between Adaptive and Non Adaptive Models

There is a perceived impression that adaptive models will cause a loss of compression in the adaptation of the source. Generally, it can be proven that there is an adaptive model that is only slightly worse than the static model [Bell, Cleary & Witten, p.56]. Conversely, it can be shown that a static model can be arbitrarily worse than an adaptive model.

Proposition that adaptive models can only be a little worse than the best non adaptive model: [Bell, Cleary & Witten, p.56]

> *Given a set of conditioning classes with K members, an alphabet of q symbols and a message length of n, there are models that will take at most* $Kq\log n$ *bits more than the best possible non adaptive model.*

The difference $Kq\log n$ is approximately the number of bits required to represent the model. This means that in a semi-adaptive model (where the model must be sent with the message), the same number of bits will be used for the adaptive model.

Corollary:

There is an adaptive model where the compressed string will be at most $\frac{(q+1)}{2\log n}$ *bits longer than the original string [Bell, Cleary & Witten, p.56].*

Conversely it can be shown that for any message, the static model may arbitrarily expand it, and by the corollary above be worse than a good adaptive model.

5.2.2 Types of Models

There are many different forms of models some if which will be described below. These models may be fixed, semi-adaptive or adaptive models.

5.2.2.1 Finite State Model or Markov Model

These models are based on finite state machines (FSM's). They have a set of states together with a set of transition probabilities [Bell, Cleary & Witten, p.35]. These models are deterministic and can implement a finite context model. The probability of the entire message is the product of probabilities out of each state.

A well known model of this form is Dynamic Markov Coding (DMC) [Bell, Cleary & Witten, p.191]. Initially this model is small but grows as new states are dynamically added. Frequency counts are maintained at each node and when this transition becomes popular a new state is cloned.

5.2.2.2 Finite Context Model

Context modelling is a special case of Markov modelling. The term Markov modelling is often used to refer to context modelling. A context model predicts successive characters taking into account characters already seen. Probabilities computed of symbols will differ depending on the symbol that precedes it, i.e., the probability of the incoming symbol is calculated based on the context in which the symbol appears. These models are usually implemented by finite state machines.

Well known finite context models are the *p*rediction by *p*artial *m*atching (PPM) models. There are many variants of PPM. Ideally one would like to use the highest order model possible, but initially there is not enough statistical information for efficient representation of the input source. A higher order model will give the best compression but does require more time to gather these statistics. In contrast, a lower order model will gather the statistics quicker. The solution [Cleary & Witten, p.396] is to use a partial match where a higher order model is formed, but a lower order model is used when the high order predictions are not available.

5.2.2.3 Grammar Models

In this type of model, probabilities are associated with productions in the language. Formal languages can easily be compressed but it is difficult to model a natural language.

5.2.3 Channel Error Recovery

With effective data compression, no redundancy remains in the compressed stream to permit the receiver to recover synchronisation once it has been lost. As part of the modelling process, both the encoder and decoder model must maintain synchronisation. Any form of error will cause the decode process to incorrectly update the model. It is the responsibility of the channel to provide an error free transport of the bit stream.

This loss of synchronisation was thought to increase the security of a cipher [Witten & Cleary, p.400]. However, it was shown that this added no additional security to a cipher [Boyd 1992]. Other researchers showed that the keyspace was only as large as the model. *(This paragraph will be explained in Chapter 6 on cryptography basics).*

A controlled amount of redundancy may be added to arithmetic coding [Boyd, Cleary, Irvine, Rinsma-Melchert & Witten] by adding an additional error symbol. When the decoder detects this error symbol it knows that an error has occurred.

5.3 Conclusion

This chapter has reviewed some of the more common modelling ideas. There is currently much research in this area, especially with variants of PPM.

Chapter 6

CRYPTOGRAPHY

6.1 Introduction

Cryptography deals with the transformation of a plaintext message into a reversible ciphertext message by encryption. Recovery is done by decryption of the ciphertext. The aim of cryptography is the keep the plaintext secret from the enemy.

Some services that are offered by cryptographic systems are [Haykin, p.816]:

- *Secrecy*, which denies unauthorised access to the contents of a message.
- *Authentication*, which is the validation of the source of the message.
- *Integrity*, which ensures that the message has not been modified while in transit.
- *Non repudiation*, which ensures that the sender cannot alter the contents at a later time, or to deny transmitting the message.

6.1.1 Cryptanalysis

Cryptanalysis is the science of recovering the plaintext or the key without having access to the key [Schneier 1994, p.4]. Cryptanalysis is achieved by determining the key from the ciphertext and the *a priori* probabilities of the various plaintext and keys. This is known as an attack. It is always assumed that the cryptanalyst has full details of the encryption algorithm[25]. An attack may either be passive or active. The cryptanalyst will monitor the cipher channel in a passive attack. The active attack will attempt to alter data on the cipher channel.

The type and form of the cryptanalytic attack is important when the security of any cryptosystem is being considered. The general types of cryptanalytic attacks are [Haykin, p.819]:

[25] Kerckhoff's Principle: "*The security of a cryptosystem must not depend on keeping secret the crypto algorithm. The security depends only on keeping secret the key*". [Beutelspacher, p.15]

1. *Ciphertext-only attack.* The cryptanalyst only has the ciphertext from several messages all encrypted with the same algorithm. Use is made of the statistical knowledge of the structure of a language.

2. *Known-plaintext attack.* The cryptanalyst has both the plaintext and the corresponding ciphertext formed by the same secret key. For example, certain plaintext pairs maybe common, i.e. a destination header on a message.

3. *Chosen-plaintext attack.* The cryptanalyst has both the plaintext and ciphertext and the ability to choose the plaintext. For example, each time the plaintext message is sent, the corresponding ciphertext message is received.

4. *Chosen-ciphertext attack.* The cryptanalyst chose the ciphertext and obtain the corresponding plaintext. This attack is mainly applicable to public key algorithms [Schneier 1995, p.7].

In general, a ciphertext-only attack can be viewed as the weakest form of attack that the cryptographic system can be subjected to. Ideally the cryptographic system must withstand both known-plaintext and chosen-plaintext attacks.

An *unconditionally secure* cipher is one in which it is impossible to recover the plaintext given all the ciphertext and computing resources [Schneier 1994, p.7]. The one time pad (OTP)[26] is of this class.

A *computationally secure* cipher is one that cannot be broken with current available resources. There is no bound paced on the amount of computation. For example, the well known ciphers; DES[27] and RSA[28] fall into this class.

6.1.2 Forms of Cryptography

Cryptographic systems that rely on the use of a single and private piece of information known as the key is referred to as secret key cipher systems. Public key cryptographic systems have two keys, one being a private key and the other known as a public key.

[26] The one time pad is a method whereby the ciphertext is generated by XORing the plaintext with a random key which is of the same length of the ciphertext. This random key is used only once and then discarded.

[27] Data Encryption Standard. DES is symmetric secret key block cipher of 64 bits and a key of 56 bits. A complete description is given in any standard text [Schneier 1995, p.270].

6.1.2.1 Secret Key

A secret key or symmetric cryptographic system is shown in Figure 5. The message source generates the plaintext, X which is encrypted into the ciphertext, Y based on the single secret key, Z. A weakness of the secret key system is that the common secret key must be passed on to the decrypter by some secure channel *(Public key systems overcome this problem).* Usually the enemy cryptanalyst will have access to the ciphertext channel as shown in the figure.

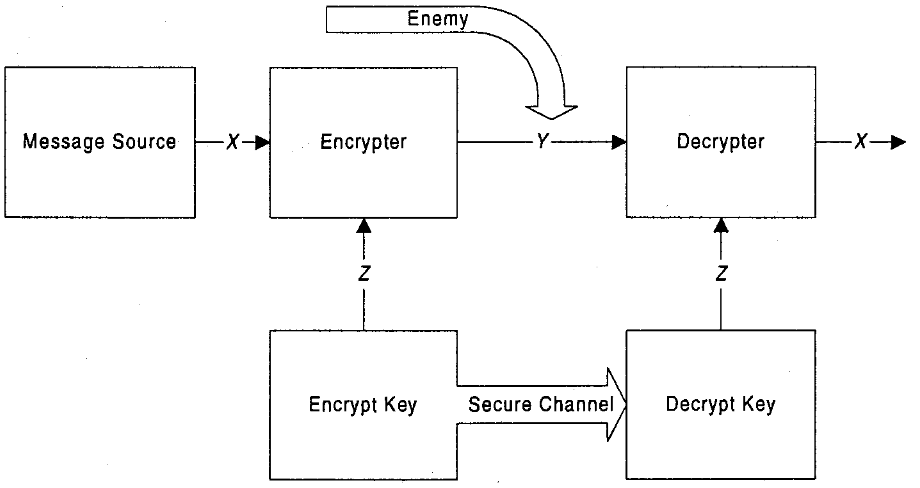


Figure 5. Secret Key Cipher System

Encryption and decryption are performed with a symmetric algorithm denoted by:

$E_Z(X) = Y$ and $D_Z(Y) = X$

Symmetric key algorithms can be of two categories:

- *Stream ciphers*, which operate on the plaintext on a bit per bit basis [Schneier 1994, p.170], e.g. RC4, video or data streams, or
- *Block ciphers*, where the bits of the plaintext are grouped into blocks [Schneier 1994, p.219], e.g. RC2, DES, IDEA, Blowfish.

[28] RSA is named after the inventors, Rivest, Shamir & Adelman. This public key cipher will be described in a later section.

6.1.2.2 Public Key

A public key algorithm uses different keys for encryption and decryption [Schneier 1994, p.273]. The encryption key is usually called the public key and the decryption key, the private key. In contrast, authentication systems use the private key to generate the signature and the public key for verification of the signature.

Public key algorithms are based on computationally hard problems. Generally the algorithms get their security from calculating discrete logarithms in a finite field or by factoring large prime numbers. The RSA algorithm [Schneier 1994, p.283] is very common and is as follows:

Public Key:

Choose n, a product of two secret primes, p and q

so that e is relatively prime to $(p-1)(q-1)$

Private Key:

Use Euclid's algorithm to compute the private key, $d = e^{-1}(\mathrm{mod}(p-1)(q-1))$

Encrypting:

$c = m^e(\bmod n)$

Decrypting:

$m = c^d(\bmod n)$

Public key algorithms are not suited to bulk encryption as their algorithms are computationally intensive. They have an encryption and decryption rate which is much slower than the symmetric methods.

6.1.3 Information Theoretic Approach

Shannon considered two notions of security [Shannon 1949]. These are:

- *Theoretical security*. The enemy cryptanalyst is assumed to have unlimited time and computing resources. As will be shown in this information theoretic approach, this leads to a pessimistic conclusion that the amount of secret key required to build a secure cipher will be impracticably large.
- *Practical security*. The cryptanalyst has limited computing resources and time. Another way to consider the problem is in terms of the value of the information. Is the value of the information less than the cost to break the cipher? Public key algorithms are intended to provide practical security.

The rest of this section will consider aspects of theoretical security.

Let $X^n = (x_1, x_2, x_3, \ldots, x_n)$ be a plaintext message of length *n*, and $Y^n = (y_1, y_2, y_3, \ldots, y_n)$ be the corresponding ciphertext message of the same length. The secret key, *Z* is based on some probability distribution. For clarity purposes, the superscript *n* will be dropped.

The uncertainty about *X* is expressed as the entropy $H(X)$, and the uncertainty about *X* given the knowledge of *Y* is the conditional entropy $H(X|Y)$.

The reduction in uncertainty is called the mutual information. The mutual information between *X* and *Y*, (measure of dependence between *X* and *Y*) is given in ***equation 1*** and defined [Cover & Thomas, p.20] by:

$$I(X;Y) = H(X) - H(X|Y) \qquad \ldots\ldots\ldots\ldots.1$$

Shannon defined this mutual information as a basic measure of secrecy.

6.13.1 Perfect Secrecy

Perfect secrecy [Shannon] is obtained when the *a posteriori* probabilities of the ciphertext are identical to the *a priori* probabilities of the plaintext. The plaintext, *X* and the ciphertext, *Y* must be statistically independent of each other. This means that the mutual information $I(X;Y) = 0 \quad \forall n$. From equation 1,

$$H(X|Y) = H(X) \qquad \ldots\ldots\ldots\ldots.2$$

Thus the cryptanalyst can only guess the plaintext *X*, given the ciphertext *Y*, according to the probability distribution of all possible messages.

Given the key *Z*, the joint probability is $H(X,Z|Y)$. From ***equation 2***,

$H(X|Y) \leq H(X,Z|Y)$ and from the Chain Rule,

$H(X|Y) = H(Z|Y) + H(X|Y,Z)$

The conditional entropy $H(X|Y) = 0$, if and only if *Y* and *Z* together uniquely determine the plaintext *X*. Hence from equation 2 and substituting for $H(X|Y,Z)$ yields:

$H(X|Y) \leq H(Z|Y)$, and because they are statistical independent,

$H(X) \leq H(Z)$

Shannon's bound for perfect secrecy is $H(Z) \geq H(X)$. With the Vernam cipher or one-time-pad, the key is as long as the plaintext, so the conditions for perfect secrecy will be met, providing that the key is random (the key sequence is comprised of statistically independent and uniformly distributed bits). This is the prime motivation for homophonic coding [Jendal, Kuhn & Massey, p.382].

6.1.3.2 Key Equivocation

Shannon defined the *key equivocation function*, or conditional entropy of the key, given the first *n* digits of ciphertext, as $f(n) = H(Z^n | Y^n)$.

The key equivocation function is a measure of the number of values of the secret key *Z* that are consistent with the first *n* digits of the ciphertext.

Because $f(n)$ can only decrease as *n* increases, Shannon called a cipher system *ideal* if $f(n)$ approaches a non-zero value as $n \to \infty$, and *strongly ideal* if $f(n)$ is constant, i.e. $H(Z|Y) = H(Z) \; \forall n$.

This implies that in a strongly ideal cipher, the ciphertext sequence is statistically independent of the secret key. This is illustrated in the figure below.

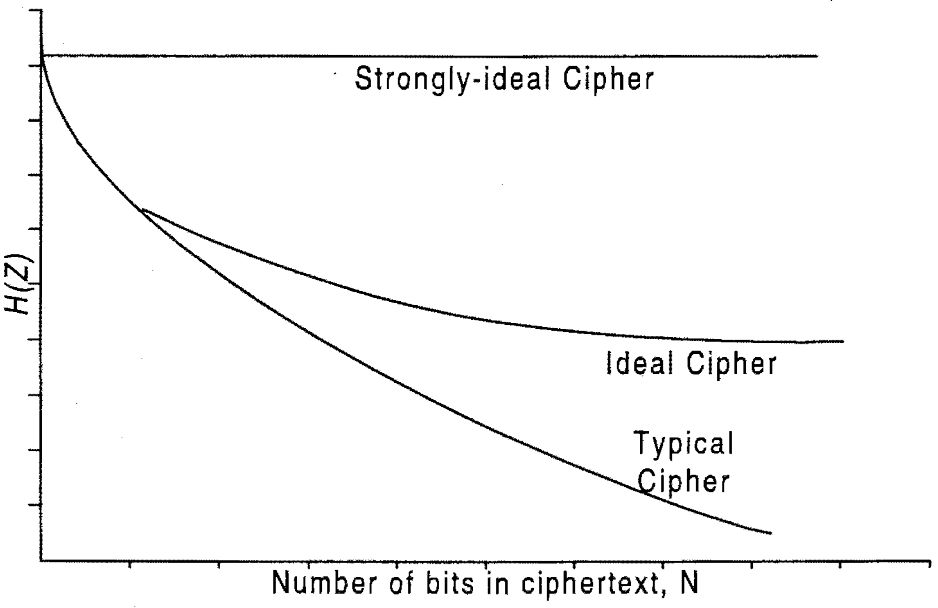


Figure 6. Key Equivocation Function

6.1.3.3 Unicity Distance

The unicity distance is an approximation of the amount of ciphertext such that the sum of the real information (entropy) in the corresponding plaintext plus the entropy of the key equals the number of ciphertext bits used [Schneier 1994, p.192]. From a theoretical point of view, the unicity distance is amount of ciphertext required for there to be only one reasonable solution for cryptanalysis.

Unicity distance is defined [Schneier 1994, p.192] as the entropy of the cryptosystem divided by the redundancy of the language, $U = \frac{H(Z)}{D}$

The rate of a language is defined as $r = \frac{H(X)}{N}$ and represents the average number of bits of information in each letter. For English the value of r ranges from 1.0 to 1.5 bits per letter.

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 $R = \log_2 L$ which is equal to the maximum entropy of the individual letters. For a 26-letter alphabet, $R = \log_2 26 = 4.7$ bits per letter.

The redundancy of a language with rate r and absolute rate R is defined as $D = R - r = \log_2 L - \frac{H(X)}{N}$

The percentage redundancy is defined as $\rho = \frac{D}{\log_2 L} = 1 - \frac{H(X)}{N \log_2 L}$

With a strongly ideal cipher system the redundancy $r = 0$, so that the unicity distance tends to infinity, as illustrated in Figure 5. This implies that the message source emits completely random plaintext, so that $H(X) = N \log_2 L$.

Unicity distance guarantees insecurity if it is too small, but does not guarantee security if it is large [Schneier 1994, p.193].

6.1.3.4 Unconditional Security

When a cipher is ideal it is not necessarily unconditionally secure. It may be unconditionally secure in a cipher-text attack but fail in a known-plaintext attack [Rueppel, p.73]. The security of an ideal cipher depends on the complete confidentiality of the plaintext. What is required to make a cipher unconditionally secure is both a perfect and strongly ideal cipher. This is the goal of homophonic coding.

6.1.4 Diffusion and Confusion

Shannon suggested two principles in the design of practical ciphers, confusion and diffusion [Schneier 1994, p.193].

- *Confusion* uses transformations that complicate the determination of how the statistics of the ciphertext depend on the statistics of the plaintext by spreading the redundancy. Confusion is achieved by substitution.
- *Diffusion* spreads the influence of a single plaintext letter over many ciphertext digits. This hides the statistical structure of the plaintext. Diffusion is achieved through transposition (also known as permutation).

6.2 Classical Cipher Methods Of Substitution And Transformation

There are many forms of ciphers and only a few will be mentioned. The presence of redundancy allowed many of the classical cipher methods to be successfully attacked [Irvine].

6.2.1 Substitution Ciphers

There are four types of classical substitution ciphers [Schneier 1994, p.8]:

1. *Mono alphabetic* or simple substitution where a character of plaintext is replaced with a corresponding character of cipher text.
2. *Homophonic* substitution in which one character of plaintext can map to one of several characters of ciphertext. For example, the symbol A, could map to any one of the set $\{1, 5, 9, 11, 30\}$. Homophonic substitution is discussed in more detail in Section 6.3 because it forms a major part of this dissertation.
3. *Polyalphabetic* substitution cipher that consists of multiple simple substitution ciphers. The cipher used by Julius Caesar[29] was of this type.

[29] Each plaintext character in the Caesar cipher is substituted with the character three to right modulo 26. (For example, character A is replaced by D) [Schneier 1995, p.11].

4. *Polygram* cipher where blocks of characters are substituted in groups. For example *ABC* could be replaced with *ZTQ*.

6.2.2 Transposition Ciphers

The characters of the plaintext remain the same but their order is changed [Schneier 1994, p.10].

6.3 Homophonic Substitution

Homophonic substitution is a one-to-many mapping of the source symbol. The term homophone means "sound the same" which implies that different codewords are assigned to the same source symbol. The idea is to flatten the frequency distribution of the ciphertext.

A homophonic substitution cipher [Welsh, p.213] encrypts a message from alphabet $\sum_1$ into a random cipher text from alphabet $\sum_2$ where $|\sum_2| \geq |\sum_1|$. There is a one-to-many mapping and the codewords from alphabet $\sum_2$ are known as homophones. A codeword from the one-to-many mapping of alphabet $\sum_2$ is picked at random to represent a symbol from alphabet $\sum_1$. There is a bandwidth expansion associated with homophonic substitution.

This form of homophonic substitution is referred to as classical or conventional homophonic substitution in this document. Homophonic substitution will be considered in more detail in the following chapters.

We make a distinction between homophonic substitution and homophonic coding. To clarify later descriptions we use the term homophonic substitution to refer to a complete cipher, that is, it produces ciphertext. Whereas with homophonic coding we imply that the output will be conditioned with a source coder to produce random plaintext which is then expected to encrypted. Homophonic coding thus conditions the plaintext.

In a theoretical sense, homophonic coding allows the achievement of ideal ciphers because it can achieve an infinite unicity distance by having zero redundancy [Boyd 1991, p.273].

6.4 The Role of Data Compression in Cryptography

Lossless data compression, before encryption, will remove some redundancy. Besides the obvious reduction in the length of the message to be encrypted, it will increase the unicity distance making the cipher somewhat more secure. If it were possible to have perfect compression and remove all redundancy, the plaintext would be completely random. Unfortunately, even though is there a negligible amount of coding overhead, there is always some form of modelling overhead [Boyd 1991, p.276].

However, the compressed output is not uniformly distributed and cannot withstand a known-plaintext or chosen-plaintext attack [Bergen & Hogan 1993].

With dictionary based compression methods, for example LZW, some plaintext forms part of the message[30] (see the LZW example, where the dictionary token can be followed by a token comprising the source language.).

Adaptive text compression does give the idea that it may provide a good cryptosystem [Boyd 1992, p.5], [Witten & Cleary] for the following reasons:

1. A single bit change in the stream changes the encoded stream.
2. Removal of redundancy implies that the encoded stream has a random appearance.
3. The adaptive model will be different for each stream encoded.
4. A large keyspace is available. The model becomes the effective key.

Boyd [Boyd 1992, p.10] showed that none of these factors below contributed to the security of the cryptosystem.

1. An error in the bit stream is a problem for adaptive compression and encryption. An error free channel is required.
2. There is a weakness in a known or chosen plaintext attack when the encryption is initiated. (The model has not yet seen sufficient input to adapt to the source).
3. A large keyspace is no guarantee of security.

Irvine showed [Irvine] the compression methods of arithmetic coding, Huffman, semi-splay tree and Ziv-Lempel all contained security flaws. Furthermore, even an adaptive arithmetic coding compression algorithm fails under a chosen-plaintext attack [Bergen & Hogan 1993]. Compression provides a deterministic mapping between the message source and the output of the compressor [Penzhorn, p.343].

[30] Note: this fact became apparent to the author in developing software for V.42 bis compression in modems.

6.5 Conclusion

This chapter has provided an introduction to the subject of cryptography and laid the groundwork for the information theoretic treatment of homophonic coding. We also showed why compression alone is not suitable for increasing the security of a cipher. Unconditional security requires both a perfect and a strongly ideal cipher.

Chapter 7

HOMOPHONIC CODING

7.1 Introduction

This chapter will discuss the theoretical aspects of homophonic coding. The previous chapter provided the necessary background and the *strongly-ideal* concept of Shannon. Four forms of homophonic substitution are apparent from the literature:

1. The classical definition as given in the previous chapter in Section 6.3.
2. Variable length homophonic coding based on Huffman codes [Günther] (to be described).
3. Arithmetic coding of homophones with dyadic (negative integer powers of two) probabilities [Penzhorn].
4. Universal or multiplexed homophonic coding [Massey] (to be described).

In practice, it is impossible to design a perfect source coding (compression) scheme [Massey, p.12/426] and thus to design a strongly ideal cipher.

The aim of homophonic coding is to convert a plaintext source into a random source. It will be shown that any secret key cipher may then be used to create a strongly ideal cipher system. The cipher is then able to withstand both cipher-text and known-plaintext attacks.

In all our discussions of homophonic coding, a homophonic coding system is never used in a standalone mode. It will always be used together with an encrypter. Figure 7 indicates how the homophonic coder is placed in relationship to the general purpose non-expanding encryptor.

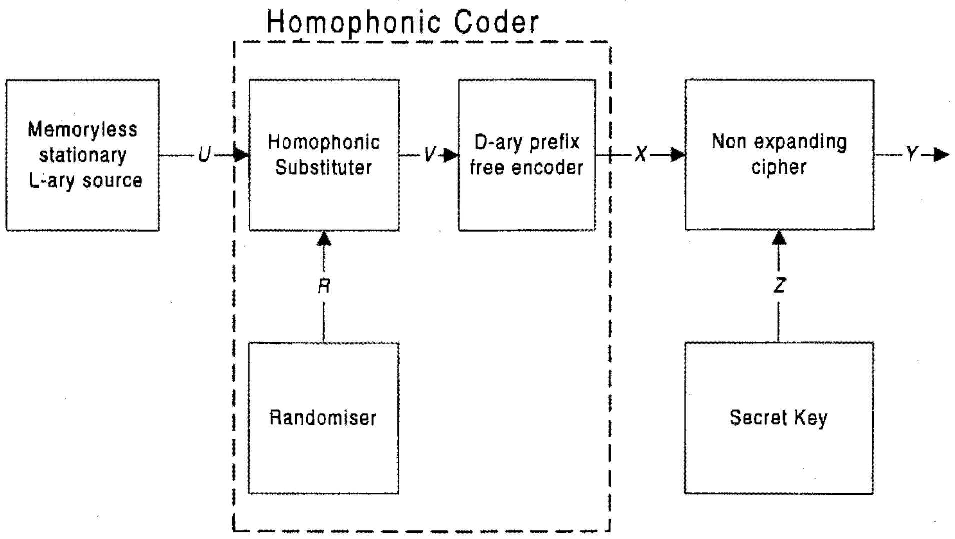


Figure 7. Homophonic Coder

7.2 Information Theoretic Analysis of Homophonic Coding

7.2.1 Background

Consider a plaintext message that produces a sequence of random variables, denoted as $U_1, U_2, U_3, \ldots$. The U_i take values in an alphabet of *L* letters, where $2 \le L < \infty$. Assume that the source is memoryless and stationary, which implies that the source sequence consists of independent and identically distributed *L*-ary random variables. Since the message source is memoryless, the coding problem for the given message source then reduces to the coding problem for the single variable $U = U_i$. The U_i are coded into a *D*-ary sequence $X_1, X_2, X_3, \ldots$ [Penzhorn, p.331]], [Jendal, Kuhn & Massey, p.385].

When $L = D^w$ for some positive integer *w* and all the values of U are equally likely, randomly assigning one of the different *D*-ary sequences of length *w* to each value of *U*, makes $X_1, X_2, X_3, \ldots$ completely random [Jendal, Kuhn & Massey, p.385]. When $D^w > L$ we have conventional homophonic substitution. Usually, *L* = *256*, *D* = *2*, *w* = *8* for an eight bit ASCII representation.

The following propositions and corollaries were cited and proved by [Jendal, Kuhn & Massey] and as such are stated here without proof, although some of proofs follow directly from the previous chapter.

Definition 1: [Jendal, Kuhn & Massey, p.383]

A cipher is non-expanding when the plaintext and the ciphertext take values from the same alphabet, so that when the key is known, the ciphertext and plaintext uniquely determine each other.

Proposition 1: [Jendal, Kuhn & Massey, p.383] & [Shannon]

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

This is readily apparent from the information theoretic treatment (Shannon's bound for perfect secrecy) in the previous chapter.

Corollary 1 to Proposition 1: [Jendal, Kuhn & Massey, p.384]

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

A non-expanding cipher has a property known as non-degeneracy [Massey]. This property will cause a change in the ciphertext when the key is changed and the plaintext is kept constant.

Corollary 2 to Proposition 1: [Jendal, Kuhn & Massey, p.384]

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

$H(X^n|Y^n) \approx H(Z)$ *for all n sufficiently large.*

Corollary 2 implies that the cipher system is unbreakable in a ciphertext-only attack when the number of possible key values is large [Jendal, Kuhn & Massey, p.384]. The cryptanalyst can only find X^n by guessing at random from the many possibilities of the secret key Z.

7.2.2 Perfect and Optimum Homophonic Coding

A homophonic scheme is perfect when the coded *D-ary* is completely random. In this dissertation we will restrict $D = 2$.

Definition 2: [Jendal, Kuhn & Massey, p.388]

A Homophonic coding scheme is perfect if the encoded binary sequence $X_1, X_2, X_3, \ldots$ *is completely random.*

Proposition 2: [Jendal, Kuhn & Massey, p.388]

For a homophonic coder as in Figure 7, the expression

$H(U) \leq H(V) \leq E[W] \log D$

holds with equality on the left if and only if the homophonic channel is deterministic, and with equality on the right if and only if the homophonic coder is perfect. Moreover, there exists a D-ary prefix-free coding of V such that the scheme is perfect if and only if $p(v)$ *is a negative integer power of D for all possible values of* v_i *of V, i.e. if and only if* $P(V = v_i) = 2^{-l_i}$ *holds for all values* v_i *of V where* l_i *is the length of the D-ary codeword assigned to* v_i.

Definition 3: [Jendal, Kuhn & Massey, p.389]

A homophonic coding scheme is called optimum if it is perfect and minimises the expected length $E[W]$ *of the binary codeword assigned to the homophonic channel output V, when the input is the message source output U.*

Proposition 2 shows that the task of designing an optimum homophonic coder is equivalent to finding a homophonic channel that minimises the entropy *V*.

Proposition 3: [Jendal, Kuhn & Massey, p.390], [Penzhorn, p.333]

A homophonic channel is optimum if and only if, for every value u of U its homophones equal (in some order) the terms in the unique dyadic decomposition

$P(U = u_i) = \sum_{j \geq 1} p_i^j$

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

Proposition 4: [Jendal, Kuhn & Massey, p.390]

For an optimum binary homophonic coder holds:

$H(U) \leq H(V) = E[W] < H(U) + 2$

Proposition 4 shows that the entropy of the input *U* of an optimum homophonic channel never increases by more than 2 bits.

Any secret key cipher system can be used for the cipher in a strongly ideal system provided that the plaintext source emits a random sequence [Massey], [Penzhorn]. This exactly what homophonic coding attempts to achieve.

7.3 Variable Length Homophonic Coding based on Huffman Coding

This form of homophonic substitution is attributed to Günther [Günther, p.407]. The homophonic coder consists of three elements: a binary memoryless message source (BMS), a homophonic channel and a Huffman source encoder.

The operation may best be explained by a simple example as given by Günther. The message source U produces symbols from the two symbol alphabet with probabilities $u_1 = \frac{1}{4}$ and $u_2 = \frac{3}{4}$. The homophonic channel then substitutes the U_i for homophones from the alphabet $V = \{v_1, v_2, v_3\}$. The selected homophones are encoded as binary codewords in the alphabet X by means of a Huffman source coder.

In this example, u_1 is always represented by v_1. Symbol u_2 is represented by a random selection of either v_2 or v_3.

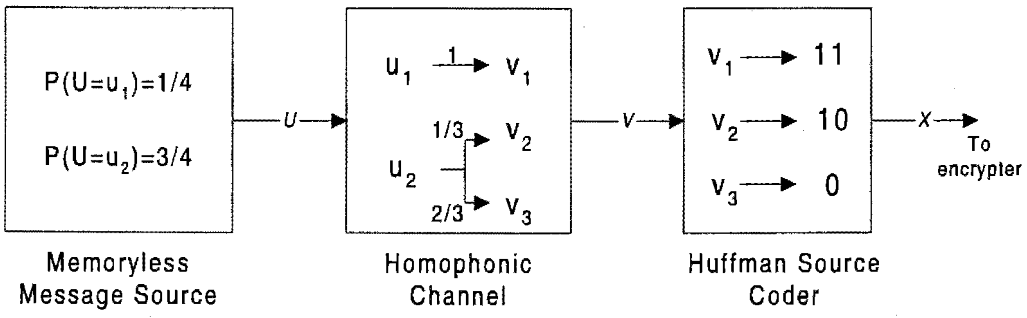


Figure 8: Variable length Homophonic Coding

This scheme cannot adapt to changing[31] source statistics [Penzhorn, p.331] and precludes its use as a practical method for creating strongly ideal ciphers. Hence the proposal to use arithmetic coding [Penzhorn].

[31] The plaintext source statistics must be known exactly.

7.4 Homophonic Coding based on Arithmetic Coding

A novel concept of using homophonic coding based on arithmetic coding was proposed by Penzhorn. This method is based on the shift and add concept of Langdon [Langdon, p.137] and is described in Chapter 4. The idea is to combine the work of Jendal, Kuhn & Massey together with the dyadic arithmetic coder of Langdon. The algorithm [Penzhorn, p.339] comprises the following steps: source modelling, design of the homophonic channel, random selection of the homophones and arithmetic coding of these homophones. Each of these steps will now be described.

7.4.1 Encoding Process

7.4.1.1 Source Modelling

The proposed implementation of Penzhorn [Penzhorn] used a fixed semi-adaptive model, but placed some limitations on the implementation. For example; scaling of the range can be undertaken after the creation of the model.

7.4.1.2 Design of the homophonic channel

The probabilities from the model are approximated by dyadic homophones. The estimated probability from the model is approximated by a series comprised of negative powers of two. This approximation process will cause a small error ε. The size of this error is reduced by increasing the word size, which will increase the number of homophones generated and thus a larger symbol alphabet.

For example, consider a symbol with a probability of 0.22. This can be dyadically approximated by
$0.22 \equiv \frac{1}{8}+\frac{1}{16}+\frac{1}{32}+\varepsilon = 0.21875+\varepsilon$

Then each of dyadic probabilities can be used as the randomly selected homophones. Using a word size of five bits, the homophones become:

$\frac{1}{8}$	00100
$\frac{1}{16}$	00010
$\frac{1}{32}$	00001

Now any one of the three words can be used to represent the symbol of probability 0.22

7.4.1.3 Random selection of the homophones

The homophones are selected by an external randomiser which introduces additional randomness into the message. This randomness accounts for the increase in entropy of 2 bits per symbol as stated in *proposition 4.*

An external randomiser using a binary symmetric source (BSS) can be used to transfer subliminal information [Penzhorn, p.341].

7.4.1.4 Arithmetic encoding of the homophones

The final step is to encode each randomly selected homophone by any arithmetic coder. In this case the shift and add arithmetic coder was proposed [Penzhorn].

7.4.2 Decoding Process

The decoding process applies the reverse coding step. It is still important that the decoder model knows the mapping of U to V. The same model must still be used with the same homophones and the range must still be re-calculated in the same manner as the encoder.

Consider the following example. The interval for symbol A is split into three homophones, A_1, A_2 & A_3. The message A_2BA_1 is sent. From the final interval, the first symbol to be decoded will be A_2, followed by B. The example in Figure 9, shows that a new range must be split up for each symbol, causing a much smaller interval that would be required with normal arithmetic coding. These results are demonstrated in Appendix B, Page 93.

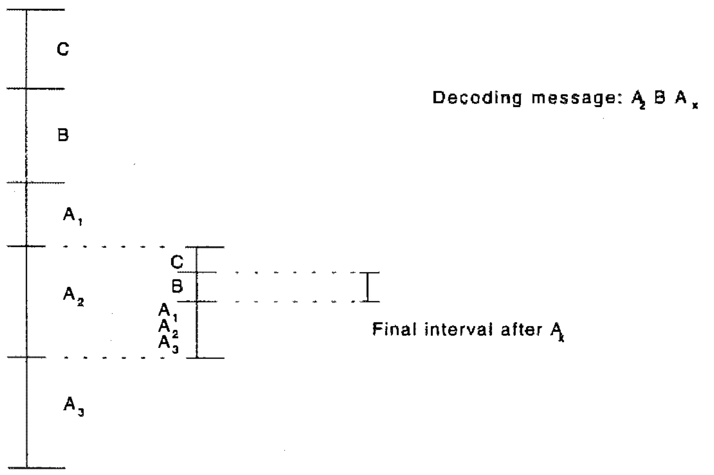


Figure 9. Decoding the final homophonic interval

7.5 Multiplexed Homophonic Coding

Massey [Massey, p.13/427] introduced a form of universal homophonic coding or generalised *plaintext stuffing* which does not require the exact knowledge of the source statistics. This method comprises a multiplexor which multiplexes the data between the message source and a BSS. The output of the multiplexor is compressed and then encrypted with any non expanding cipher as shown in Figure 10.

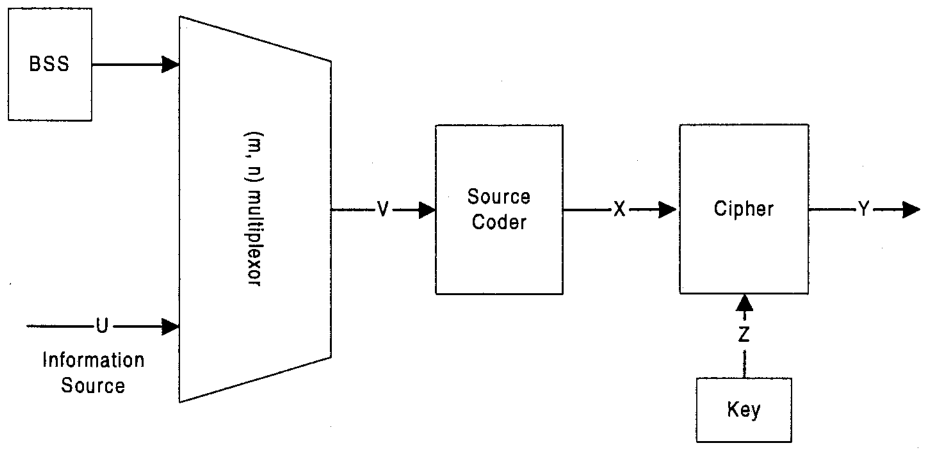


Figure 10. Homophonic coder based on a 2 input multiplexor

This system operates by outputting m random bits from the BSS followed by n digits from the message source. The sequence then repeats. The universal homophonic output is achieved by compressing the multiplexed stream V with any universal source coder.

The information is recovered by reversing the compression. Since m and n are known, the m random bits can be discarded and the n bits will represent the message. Even with a discrete stationary and ergodic input source, the multiplexor introduces non-stationarity. This means that the output will only be stationary with multiples of $m+n$ [Massey, p.12/427].

7.6 Conclusion

Homophonic coding converts a source having a non uniformly distributed alphabet into one that is uniformly distributed. Unicity distance is increased by reducing the redundancy of the language. This is achieved by data compression and homophonic coding. Homophonic coding is necessary as it impossible to have perfect source compression. Perfect compression converts the output into a BSS [Massey, p.12/426].

Furthermore, homophonic coding together with an encryption system will provide security against a known or chosen plaintext attack because it provides a probabilistic mapping between the message source and the output of the source coder [Penzhorn, p.343].

Chapter 8

UNIVERSAL HOMOPHONIC CODING

8.1 Introduction

In the previous chapter we considered a number of different homophonic coding paradigms. One major disadvantage of these methods (with the exception of plaintext stuffing of Massey) was that they were not universal. The aim of this chapter is to discuss such a universal homophonic coding system.

Penzhorn showed [Penzhorn] that a homophonic coder can be based on a number of separate entities:

- A model,
- The homophonic channel, which contains the randomiser and homophone generator,
- An arithmetic coder,
- Non expanding encryptor,
- Communication channel.

To make this homophonic coder universal, the source statistics must be adaptively modelled. In practice this system must be able to follow any message source.

We consider two forms of universal homophonic coding. One is based on a conventional method and the other on the dyadic probabilities of Penzhorn.

8.2 Conventional Universal Homophonic Coder based on Arithmetic Coding

In the conventional homophonic coding system of Figure 11, the alphabet is expanded to a desired number of homophones per input character, of the input sequence U. The randomiser will randomly select one of these expanded homophones v_i, from the sequence V. The effect of the good randomiser is to flatten the probability distribution of each character by making each selection of the input character equally likely. The encoder will encode this new symbol v_i, and the model will also assign new probabilities to this symbol v_i. The decoder will receive this new symbol and because it knows how may homophones are related to each input character it will be able to convert this symbol to a character.

There is an obvious loss in compression with this scheme due to the bandwidth expansion of the one-to-many mapping. There will be a point at which the number of homophones selected will cause an expansion of the original message. The output, however, will still be random.

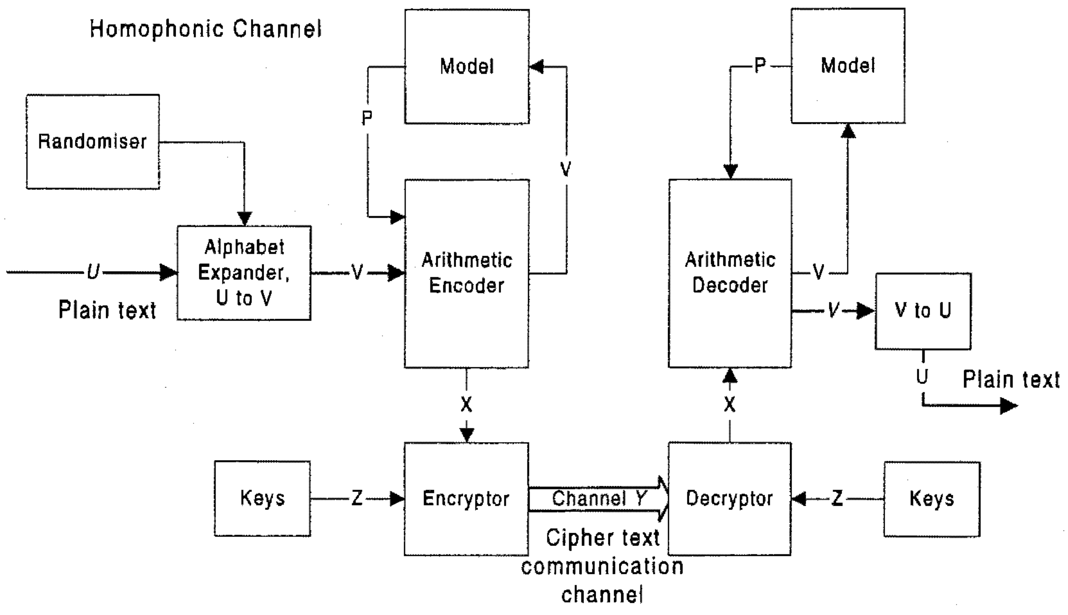


Figure 11. Conventional Universal Homophonic Coder based on Arithmetic coding

The alphabet expander can dynamically assign a set of homophones based on the probability of u_i. This means that there will be a different number of homophones for each u_i, so that the overall frequency distribution can be flattened. This dynamic assigned of homophones is not considered in this implementation and the term universal applies to fact that the model will adapt to the source statistics.

8.3 Dyadic Probability Universal Homophonic Coder

The dyadic or split probability homophonic coder shown in Figure 12 is based on the concept described by Penzhorn [Penzhorn] on page 61 of Section 7.4, Chapter 7.

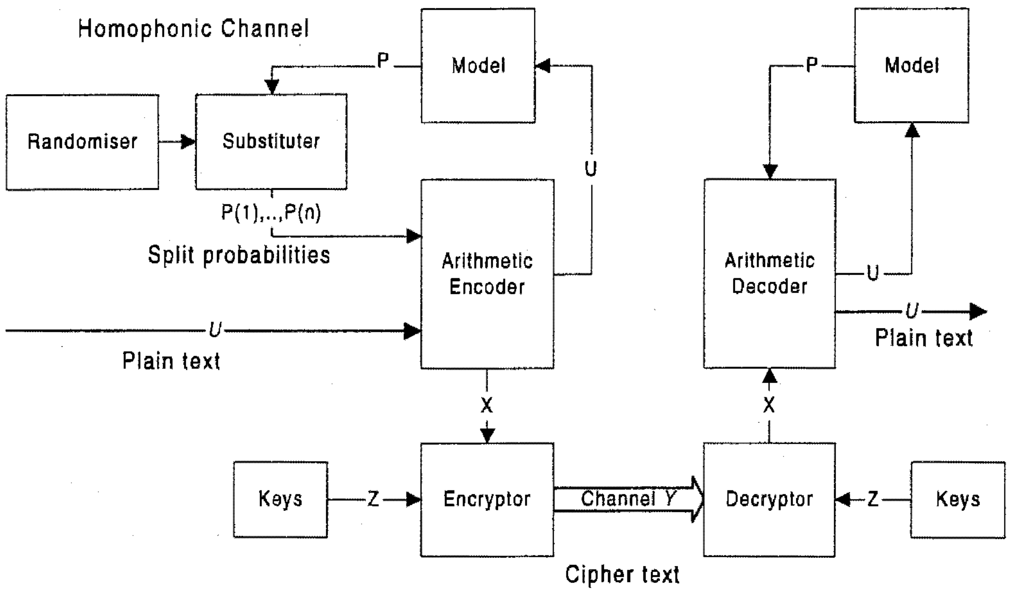


Figure 12. A Universal Homophonic Coder based on Arithmetic Coding with dyadic probabilities.

8.3.1.1 Source Modelling

An order-0 model will be used to simplify the explanation. A higher order model may be used with an associated increase in complexity. Initially an order-0 adaptive model will have all symbol probabilities set to unity, that is; all symbols have the same probability. No homophonic coding is necessary as distribution is equal, i.e. all homophones have probability of 1.

When a new symbol is processed, the count associated with this symbol is incremented and the cumulative probability increased as well. The homophonic channel will use this information to update the homophones of this symbol.

The model will also organise all the symbols so that those with the highest probabilities will always be first in the search order. The nature of locality of reference will cause these more frequent symbols to be accessed more often.

8.3.1.2 Design of the homophonic channel

The design of the homophonic channel is closely coupled with the model. Each symbol has a number of homophones associated with it. Initially with an adaptive order-0 model there will only be one homophone with a unity probability.

When a new symbol occurs and the model updates the probability, the homophonic channel will assign a new homophone to that symbol. At the same time it will update a cumulative probability entry for this new homophone.

Because the arithmetic coder cannot use an updating homophone for shrinking the interval, the model will update a standard model. After an update count is exceeded, the homophones are updated based on the current state of the model and the process is repeated until there are no further input symbols. The arithmetic decoder must keep track of the same set of homophones.

8.3.1.3 Random selection of the homophones

A random selection of one of the homophones associated with the symbol is required. This random symbol is passed to the arithmetic coding function.

8.3.1.4 Arithmetic encoding of the homophones

As long as the probability and cumulative probability for a symbol are updated correctly, any standard arithmetic coder may be used.

Arithmetic coding allows dual homophone selection. The encoder always calculates the interval from the coding step:

$$\begin{aligned}
&range = high - low \\
&high \quad = low + range \times cummulative\ probability[upper] \\
&low \quad\;\, = low + range \times cummulative\ probability[lower]
\end{aligned}$$

The upper cumulative probability is given by the random homophone. The lower cumulative probability is usually given by the previous symbol.

8.3.1.5 Non expanding cipher

Any non expanding cipher is suitable. This implementation is based on the IDEA block cipher.

8.3.2 Decoding Process

The decoding process is much simpler than the encoding process as there is no random selection of the homophones. The compressed and encrypted bit stream is first passed to the block cipher which decrypts the stream. The stream can now be passed to a matching arithmetic decoder based on the same model as the encoder. This model must maintain the same homophones as that of the encoding model.

8.4 Conclusion

This chapter explained the concept of a universal homophonic coder based on an adaptive order-0 arithmetic coder. These systems will be implemented and described in the chapter on implementation.

Chapter 9

IMPLEMENTATION

9.1 Introduction

This chapter describes the implementation of the universal homophonic coder. Both universal homophonic coders will be implemented. All implementation is based on the C programming language.

9.2 Implementation

After many attempts and deliberation, the arithmetic coder of [Bell, Cleary & Witten, p.133], [Witten, Neal & Cleary] was selected. The reason being that there is a well described implementation which has solved many of the problems related to arithmetic coding. The interface to the model is loosely coupled and thus the model and coder sections are independent of each other. The shift and add method described by Langdon could be used with the modifications by Rubin. However, it was deemed that this increased the complexity of the method. It is to this general implementation of [Bell, Cleary & Witten, p.133], [Witten, Neal & Cleary] that the homophonic coder is added.

All the arithmetic coding methods based on the Bell, Cleary & Witten [Bell, Cleary & Witten, p.133], [Nelson, p.148] and the DCC95 code suite[32] [Bloom] use a common method of interfacing with the model.

9.2.1 Interfacing the Homophonic Coder and the Arithmetic Coder

The following common routines of arithmetic encoder and decoder cleanly interface the model and coder. All functions names are based on those of Bell, Cleary, Witten & Neal.

1. *start_model().* This functions assigns the initial probabilities and the translation tables between the input characters and symbols.

[32] Data Compression Conference (Snowbird, Utah 1995).

2. *update_model(symbol).* This function updates the frequency counts for the new symbol and sorts the cumulative frequencies into ascending order to increase coding speed by ordering the translation tables between the input characters and symbols.

3. *encode_symbol(symbol, cum_freq).* This function applies the standard coding step and outputs the bits.

4. *decode_symbol(cum_freq).* This function applies the standard coding step and returns the new symbol from the input stream.

This implementation will not alter these functions but add new and similar functions for adaptive homophonic coding. Only the *main* calling routine will be adapted to include these new functions.

9.2.2 Conventional Universal Homophonic Coding

This implementation requires a small modification of the memory allocated for the cumulative frequency, frequency, index to character and character to symbol arrays. The following simple changes implement this method:

```
#define Num_of_chars  256 * NumberOfHomophones

InputCharacter = (ch * NumberOfHomophones) + random(NumberOfHomophones);

symbol         = char_to_index[InputCharacter];
```

The rest of the code remains unchanged. The *InputCharacter* is now converted to symbol before it is encoded with *encode_symbol(symbol, cum_freq).* A major advantage of this method is that the model need not be altered and any model may be used.

The decoding side is equally trivial. Each decoded symbol is divided by the number of homophones.

9.2.3 Universal Dyadic Homophonic Coding

The source code for this method can be found in Appendix C. Figure 13 indicates the data flow (without showing termination of the message). Initially the homophonic coder operates without outputting homophones until the model has some input statistics.

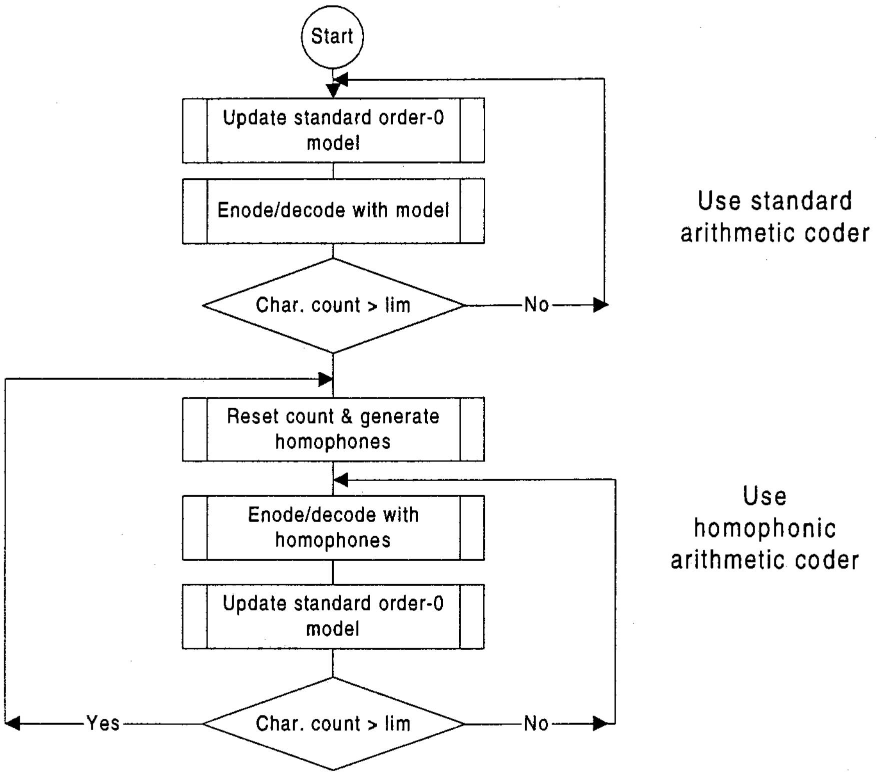

Start
Update standard order-0 model
Enode/decode with model
Char. count > lim
No
Use standard arithmetic coder
Reset count & generate homophones
Enode/decode with homophones
Update standard order-0 model
Char. count > lim
Yes
No
Use homophonic arithmetic coder

Figure 13. Universal homophonic coder flow diagram

9.2.3.1 Model

An order-0 model was selected. The main reason for using a low order model was to make the implementation and testing of the distribution of the homophones much easier. Any modelling overhead should be detectable as redundancy will be added by the homophonic coding. It would also require less memory. The order-0 model has all probabilities initially set to unity.

InitModelHC(). This function is similar to *start_model()* and does the initialisation.

CreateHomophonicModel(). This function is similar to *update_model(symbol, character)* and maintains the adaptive homophonic model. This model can be updated for every input character. This does cause a considerable overhead as the frequencies must be scaled and then for each symbol a new set of homophones must be generated. To alleviate this problem, homophones can be updated after a certain number of input characters have been processed. For this reason two models are maintained, one being the standard order-0 model with the *update_model* function call and the second being the homophonic model.

This function will call the necessary functions *ScaleToBinary()* and *GenerateHomophones(symbol)* to build up a list of homophones.

9.2.3.2 Selecting the Homophones

Selecting the homophones is trivial in the test implementation. No BSS was used. The *random* function of the compiler was used. The fixed compile time parameter *NumOfHomophones* is the argument to the random function which in turn returns a pseudo random number which can be used as an index to select a new homophone.

9.2.3.3 Homophonic Coder

EncodeCharacter(InputCharacter). This functions takes a character from the input stream and finds the associated symbol based on the character. It will now call the standard encode function *encode_symbol(symbol, cum_freq).*

9.2.4 Cryptosystem

The symmetric key block cipher IDEA was used. This cipher was previously in randomness testing [Stevens 1994]. The final results will not include the cryptosystem as the focus of this dissertation is on the conditioned plaintext.

9.3 Implementation Problems

9.3.1 Complexity of the Model

With conventional homophonic coding, the model is very similar to the model of the base implementation. In contrast, dyadic homophonic coding requires an additional model to store the working set of homophones and frequency tables.

9.3.2 Efficiency

- *Arithmetic coding.* No loss in efficiency was detected in the encoder functions are they were unchanged.

- *Homophone selection.* Scaling the frequency counts to a binary power and then splitting the scaled count into dyadic homophones increases the workload.

- *Updating the homophonic model.* Model update is the major loss in efficiency as it must maintain a new set of cumulative frequencies based on the homophone selection.

9.3.3 Memory Requirements

Homophonic coding consumes a large amount of memory. With an order-0 model, enough memory must be allocated for every homophone to be associated with each character. This means that for, say a 256 symbol alphabet, the memory size in bytes is at least 256 times the number of homophones.

The dyadic implementation requires more memory than the conventional homophonic coder as separate frequency, cumulative frequency, indices and homophonic tables must be maintained.

9.4 Conclusion

A conventional homophonic coder using arithmetic coding was successfully implemented. Although the model was restricted to an order-0, this was purely for ease of the implementation and the testing phase. The PPM models could be adapted or even the adaptive dependency model [Abrahamson]. This is an order-1 model designed to be implemented with the same arithmetic coding routines of Bell, Cleary, Witten & Neal.

Chapter 10

RESULTS

10.1 Introduction

The chapter presents the results of the universal homophonic coder described in the previous chapters.

10.2 Randomness Tests

A suite of tests to detect non randomness was developed and used to report the following results [Stevens 1994]. The source code is available at: ftp.ox.ac.uk /pub/cryptanalysis/alltest.tar.gz

These tests are the frequency, serial, runs, poker and Maurer's bit test. The randomness test result will indicate either a pass or fail result. A pass indicates that all the tests do not detect any non randomness. (The actual test results were not included in this dissertation and they would considerably lengthen this document.) The results are shown in the following tables and are tabulated in columns to indicate the type and size of the original source file, the resultant file size and whether non randomness was found:

- Standard implementation of an arithmetic coder.
- Homophonic channel together with the arithmetic coder.

10.2.1 Results of Conventional Homophonic Coding

The number of homophones per character was changed to note the difference in compression and to detect any bias in the conditioned plaintext. Table 2 indicates these results. The random number generator is of the linear congruential type and part of the compiler library. Although this generator is known to be poor for cryptographic purposes, it did not affect the non "randomness".

Table 2: Test results for an adaptive arithmetic and conventional homophonic coders

TYPE OF FILE	ORIGINAL FILE		STANDARD ARITHMETIC CODER		CONVENTIONAL HOMOPHONIC & ARITHMETIC CODER 4 homophones/char		16 homophones/char	
	SIZE (Bytes)	Randomness	SIZE (Bytes)	Randomness	SIZE (Bytes)	Randomness	SIZE (Bytes)	Randomness
Text	8568	Fail	5233	Fail	7655	Pass	10379	Pass
Text	243862	Fail	150164	Fail	213513	Pass	282964	Pass
Binary	58616	Fail	38676	Fail	53480	Pass	69247	Pass

Note how compression is reduced with the homophonic coding and even expanded when the number of homophones is increased. This is to be expected as this is not an optimum homophonic coder. Therefore, the entropy will be increased by more than 2 bits per symbol as $D^{w} > L$.

10.2.2 Results of Dyadic Homophonic Coding

Table 3: Test results for an adaptive arithmetic and dyadic probability homophonic coders

TYPE OF FILE	ORIGINAL FILE		STANDARD ARITHMETIC CODER		HOMOPHONIC ARITHMETIC CODER	
	SIZE (Bytes)	Randomness	SIZE (Bytes)	Randomness	SIZE (Bytes)	Randomness
Text	8568	Fail	5233	Fail	10231	Fail
Text	243862	Fail	150164	Fail	283434	Fail
Binary	58616	Fail	38676	Fail	71308	Fail

The results of dyadic universal homophonic coding were not satisfactory. Generally all files were significantly expanded in contrast to the theory. Consider the text file of 243863 bytes in length. An order-0 model reduced this to 150164 bytes. From the theory, we expected an expansion of not more than two bits per symbol. This theory would then have predicted a file with a length of not more than 211130 bytes[33]. In our case, the file was expanded to 283434 bytes, considerably more.

The expansion is to be expected in arithmetic coding as smaller ranges are used to encode a symbol. This causes the overall range to narrow considerably and thus require more bits for its representation. The theory of Jendal, Kuhn & Massey state in Proposition 2 (Page 59) than there will be an expansion of no more than 2 bits per symbol. This proposition only applies to prefix-free codes. Although arithmetic coding produces an uniquely decodable code, it is not prefix-free.

Although the number of ones and zeroes were close, the runs test and Maurer's test failed the sequence indicating some form of non randomness.

10.3 Known Plaintext Attacks

These tests are complex and were not considered in the scope of this dissertation.

10.4 Conclusion

Conventional universal homophonic coding performed as expected. There was a loss in compression related to introduction of multiple symbols and the associated loss in stationarity. Although this implementation of a conventional homophonic coder suffered from considerable modelling overhead, the *randomness* test results were satisfactory.

Universal dyadic homophonic coding did not perform as expected from the theory, but did perform as expected in terms of arithmetic coding.

We did not include the randomness results with the IDEA cryptosystem. The reason for this is that we are primarily interested in the randomness of the homophonic coder.

$150164 \times 8 = 1201312\,bits$

[33] $243862 \times 2 = 487724\ bits\ as\ per\ theory$

$\lceil (1201312 + 487724) \div 8 = 211130 \rceil bytes$

Chapter 11

CONCLUSION

11.1 Hypotheses

In Section 1.3, two hypotheses were made. Based on the research undertaken, the following conclusions about these hypotheses can be made:

1 a) The conventional homophonic coder which was implemented and tested, produced satisfactory results. The results of Section 10.2.1 clearly show that universal conventional homophonic coding can indeed be implemented together with arithmetic coding.

1 b) The implementation of universal dyadic homophonic coding based on arithmetic coding did not produce satisfactory results.

- The output was non random indicating a cryptological weakness, and
- The output was considerably expanded in contrast to the theory on of Jendal, Kuhn and Massey (Proposition 4 on page 59).

The reasons for the poor results will be given in the following section (11.3).

2. We did not uphold our second hypothesis, that no special form of arithmetic decoder was required for homophonic coding based on arithmetic coding. This was proved in the test implementation and proved by simple arithmetic (given on Page 90) based on the decoding inequality [Witten, Neal & Cleary, p.539]. The decoder model of a homophonic coder based on arithmetic coding must always follow the encoder model and use exactly the same homophones.

   The concept of universal dyadic homophonic coding appears to be a feasible implementation when discussing arithmetic coding (see Page 34). But as in author's case, the concept of symbol probabilities and coding interval was thought to be the same. This is not the case.

11.2 Objectives

A number of objectives were considered in this research. Data compression and cryptography were investigated and used as introductory chapters for this document. Randomness was investigated and a suite of tests was developed for use in this research. Various forms of universal homophonic coders based on arithmetic coding were implemented and their results evaluated. The terms of *homophonic substitution* and *homophonic coding* were defined to alleviate confusion between them.

11.3 Reasons for the results of universal dyadic homophonic coding

This implementation of universal dyadic homophonic coding does not function as expected and produces considerable expansion of the output in comparison to the original message. Furthermore, it does not make the output more random. The following reasons are apparent:

- Arithmetic coding reduces the range for symbol with smaller probabilities. When a symbol, for example, $u_i = \frac{3}{4}$ is converted into dyadic homophones, say $v_{i1} = \frac{1}{2}$ and $v_{i2} = \frac{1}{4}$, both of these homophones now have a lower probability. When $v_{i2} = \frac{1}{4}$ is coded, the range is reduced and when another homophone is used, the arithmetic coder will use this previous range and further reduce this already small interval. The arithmetic coder now requires more bits to represent is this smaller interval. Hence, the considerable data expansion. Furthermore, the very small interval eventually causes representation problems with fixed word length implementations.

- *Proposition 2* [Jendal, Kuhn & Massey, p.388] states $H(U) \leq H(V) \leq E[W] \log D$. This equation applies to prefix free codes which implies that each *distinct* *D*-ary sequence is a codeword [Jendal, Kuhn & Massey, p.389]. Although arithmetic coding produces an uniquely decodable code, it is not a prefix free code [Howard & Vitter 1992, p.86]. The statement that no more than two bits will be added onto the entropy of the symbol, applies to prefix free codes. Therefore, the result of *Proposition 4* [Jendal, Kuhn & Massey, p.390] (page 59 of this dissertation) may not apply to this implementation.

11.4 Is Universal Homophonic Coding worthwhile?

Penzhorn [Penzhorn, p.343] stated that a combination of homophonic and arithmetic coding did provide security against a known-plaintext and a chosen-plaintext attack when used together with a non expanding cipher. Compression can only provide a deterministic mapping between the plaintext and ciphertext whereas homophonic coding provides a probabilistic mapping.

Homophonic coding causes a loss in compression due to the addition of randomness and the modelling overhead. It is expensive in terms of memory and computing resources. Boyd [Boyd 1991, p.276] stated "*equally other methods of adding randomness may be just as effective*". This statement is justified based on the results of this dissertation. It may be more effective to have conventional adaptive data compression with a high order model followed by a stream cipher before the block enciphering process. This super-enciphering mechanism is used in practice, e.g., DVB scrambling of a MPEG 2 compressed video stream.

11.5 Future Research

During this research into universal homophonic coding, many concepts for improvements and future investigation were evident. These include the areas of:

- *Formal mathematical proof of why dyadic homophonic coding with arithmetic coding does not adhere to the theory of Massey, Jendal & Kuhn.*

- *Cryptanalysis.* One can never prove complete security, however, it is necessary that some form of attack be undertaken on the homophonic coder and cipher system.

- *Fast coding.* The range of the arithmetic coder can be constrained to binary powers. This will eliminate some of the divide operations.

- *Modelling.* Use of a higher order of modelling. This is quite memory intensive as by nature homophonic coding by itself does consume large amounts of memory. As previously mentioned, the addition of complex model to a conventional homophonic coder based on arithmetic coding should not present a problem besides the memory requirements.

- *Dynamically assigning homophones in a conventional homophonic coder.* Vary the number of homophones per character to ensure a flat probability distribution. Even though our results pass the standard non-randomness tests, flattening the distribution should make cryptanalysis more difficult.

- *Use of conventional homophonic coding when initialising the model.* When the adaptive model is initialised, all the probabilities are the same. While statistics are being gathered, conventional homophonic coding can be used until the model has sufficient statistical data to calculate the homophones. A fixed model tends to have repeating binary patterns [Bergen & Hogan 1992].

- *Memory requirements.* This research is closely associated with the model and form of data structure selected. There would be some processing speed in dynamically allocating memory. This may resolve some of the memory requirements.
- *Data structures.* Select an efficient form of a data structure. Some programming languages are inefficient in accessing[34] members of a structure.
- *Subliminal transfer of information.* The random homophone selection can be used as a subliminal channel to transfer other information on top of the homophonic channel. The decoder function is no longer trivial.

[34] C uses multiplication to access elements in a structure.

BIBLIOGRAPHY

Abrahamson, D.M.; An Adaptive Dependency Model Source Model for Data Compression. *Communications of the ACM*, Vol. 32, No. 1, January 1989, p.77-83.

Beker H. & Piper, F. 1982; *Cipher Systems: The Protection of Communications.* John Wiley & Sons 1982.

Bell, T.C. Cleary, J.G. & Witten, I.H. 1989; *Text Compression.* Prentice Hall 1989.

Bergen, H.A. & Hogan, J.M. 1992; Data Security in a Fixed-Model Arithmetic Coding Compression Algorithm. *Computers & Security*, No. 11, 1992, p.445-461.

Bergen, H.A. & Hogan, J.M. 1993; A chosen plaintext attack on adaptive arithmetic coding compression algorithm. *Computers & Security*, No. 12, 1993, p.157-167.

Beutelspacher, A. 1994; *Cryptology.* Mathematical Association of America 1994.

Bloom, C.; http://vmswww.utexas.edu, and private correspondence.

Boyd, C. 1991; Enhancing Secrecy by Data Compression: Theoretical and Practical Aspects. *Advances in Cryptology - Eurocrypt 91 Lecture Notes in Computer Science.* Menezes, A.J. & Vanstone, S.A. (Eds.) Springer & Verlag 1991, p.266-280.

Boyd, C. 1992; Applying Compression Coding in Cryptography. *Cryptography and Coding II.* Mitchell C. (Ed.) Claredon Press, Oxford 1992.

Boyd C., Cleary, J., Irvine, S.A., Rinsma-Melchert, I. & Witten I; *Integrating Error Detection into Arithmetic Coding.* Technical Report, University of Waikato 1995, ftp.cs.waikato.ac.nz

Carroll, J.M. & Robbins, L.E.; Using Binary Derivatives To Test An Enhancement of DES. *Cryptologia XII*, No. 4, October 1988, p.193-208.

Chaiten, G.J.; *Algorithmic Information Theory.* Cambridge University Press, Cambridge, 1987.

Clarke, G.M. & Cooke, D. 1992; *A Basic Course in Statistics 3rd Edition.* Edward Arnold, 1992.

Cleary, J., Irvine, S.A. & Rinsma-Melchert, I.; On the insecurity of arithmetic coding. *Computers & Security*, No. 14, 1995.

Cleary, J.G. & Witten, I.H.; Data Compression Using Adaptive Coding and Partial String Matching. *IEEE Transactions on Communications*, Vol. COM-32, No. 4, April 1984.

Cover, T.M. & Thomas, J.A. 1991; *Elements of Information Theory*. John Wiley & Sons.

Ekstrand, W. 1994; *Software Implementation of Arithmetic Coders*. Masters Thesis.

Feder, M., Merhav, N. & Gutman, M.; Reflections on "A Universal Prediction of Individual Sequences". *IEEE Information Theory Society Newsletter*, Vol. 44, No. 1, March 1994.

Fenwick, P. 1994; A New Data Structure for Cumulative Frequency Tables. *Software-Practice and Experience*, Vol. 24, No. 3, March 1994, p.327-336.

Fenwick, P. 1996; Block Sorting Text Compression. *Proceedings of the 19th Australasian Computer Science Conference*, Melbourne, Australia. 1996.

Goldreich, O. & Krawczyk, H.; Sparse Pseudorandom Distributions. *Advances in Cryptology Crypto 90 Lecture Notes in Computer Science 537*. Menezes, A.J. & Vanstone, S.A. (Eds.) Springer & Verlag 1991.

Günther, Ch. G.; A Universal Algorithm for Homophonic Coding. *Advances in Cryptology - Eurocrypt 88 Lecture Notes in Computer Science No. 330*. Springer & Verlag 1988, p.405-414.

Gustafson, H., Dawson, E. & Caelli, B.; Applying Randomness Tests to Commercial Level Block Ciphers. *Advances in Cryptology Crypto 90 Lecture Notes in Computer Science 537*. Menezes, A.J. & Vanstone, S.A. (Eds.) Springer & Verlag 1991.

Haykin, S. 1994; *Communication Systems 3rd ed.* John Wiley & Sons 1994.

Hirschberg, D.S. & Lelewer, D.A. 1992; Context Modeling for Test Compression. *Image and Text Compression*. Storer, J.A. (Ed) Kluwer Academic Publishers 1992.

Howard, P.G. & Vitter, J.C. 1994; Arithmetic Coding for Data Compression. *Proceedings of the IEEE*, Vol. 82, No. 6, June 1994.

Howard, P.G. & Vitter, J.C. 1992; Practical Implementations of Arithmetic Coding . *Image and Text Compression.* Storer, J.A. (Ed) Kluwer Academic Publishers 1992.

Irvine, S.A., Cleary, J. & Rinsma-Melchert, I.; *The Subset Sum Problem and Arithmetic Coding.* Technical Report, University of Waikato 1995, ftp.cs.waikato.ac.nz

Irvine, S.A. 1996; *PhD Thesis.* University of Waikato 1996, http://www.cs.waikato/~sirvine

Jendal, H., Kuhn, Y, & Massey, J. An Information-Theoretic Treatment of Homophonic Substitution. *Advances in Cryptology - Eurocrypt 89 Lecture Notes in Computer Science 434.* Quisquater, J. & Vandewalle, J. (Eds.) Springer & Verlag 1990, p.382-394.

Knuth, D.E.; *The Art of Computer Programming* Vol. 2 *Seminumerical Algorithms,* 2nd ed. Addison-Wesley, 1981.

Lagarias, J.C.; Pseudorandom Number Generators in Cryptography and Number Theory. *Proceedings of the Symposia in Applied Mathematics,* Vol. 42, 1990.

Langdon, G.G. 1984; An Introduction to Arithmetic Coding. *IBM Journal of Researd & Development.* Vol. 28 No. 2, March 1984 p.136-149.

Leung, A.K. & Tavares, S.E.; Sequence Complexity as a Test for Cryptographic Systems, *Advances in Cryptology, Crypto 84 Lecture Notes in Computer Science,* p.468-474.

Lucky, R.W. 1991; *Silicon Dreams: Information Man and Machine.* St Martin's Press 1991.

Maurer, U.M.; Universal Statistical Test for Random Bit Generators. *Journal of Cryptology.* No. 5, 1992 p.89-105.

Massey, J.L. 1994; Some Applications of Source Coding in Cryptography. European Transactions on Telecommunications. Vol. 5, No. 4, July-August 1994, p.7/421-15/429.

Micali, S. & Schnorr, C.P.; Efficient, Perfect Random Number Generators. *Advances in Cryptology - Crypto 88 Lecture Notes in Computer Science 403.* Goldwasser, S. (Ed.) Springer & Verlag 1989.

Nelson, M. 1992; *The Data Compression Book.* M&T Books 1992.

Penzhorn, W.T. 1994; A Fast Homophonic Coding Algorithm Based on Arithmetic Coding. *Fast Software Encryption. 1994 Lecture Notes in Computer Science 1008*. Preneel, B. (Ed.) Springer & Verlag 1995.

Press W.H., Teukolsky S.A., Vettering W.T., & Flannery B.P. *Numerical Recipes in C, 2nd ed.* Cambridge University Press, 1992.

Rissanen, J. & Langdon, G.G.; Universal Modeling and Coding. *IEEE Transactions on Information Theory*, Vol. 27, No. 1, January 1981.

Rubin, F.; Arithmetic Stream Coding Using Fixed Precision Registers. *IEEE Transactions on Information Theory*, Vol. 25, No. 8, November 1979, p.672-675.

Rueppel, R.A. 1991; Stream Ciphers. *Contemporary Cryptology: The Science of Information Integrity.* Simmons G.J. (Ed.) IEEE Press, 1991.

Schneier, B. 1994; *Applied Cryptography.* John Wiley & Sons 1994.

Schneier, B. 1995; *Applied Cryptography,* 2nd ed. John Wiley & Sons 1995.

Shannon, C.E. 1948; The Mathematical Theory of Communication Systems. *Claude E. Shannon: Collected Papers.* N.J.A. Sloane & A.D. Wyner (Eds.) IEEE Press, 1993.

Shannon, C.E. 1949; The Communication Theory of Secrecy Systems. *Claude E. Shannon: Collected Papers.* N.J.A. Sloane & A.D. Wyner (Eds.) IEEE Press, 1993.

Stevens, C.C. 1992; *Modem Data Compression.* Paper presented at M&PhD Conference, UNISA. June 1992.

Stevens, C.C. 1994; *Detecting "Non-randomness" for Cryptographic Purposes.* Special Topic report COS497-W UNISA 1994.

Welch, T. 1984; A technique for high performance data compression. *IEEE Computer*, Vol. 17, No. 6, June 1984, p.8-19.

Welsh, D.; *Coding and Cryptography.* Oxford University Press 1989. (Reprint of 1988 edition, Claredon Press, Oxford).

Williams, R.N. 1991; *Adaptive Data Compression.* Kluwer Academic Publishers 1993.

Witten, I.H. & Cleary, J.G.; On the Privacy Afforded by Adaptive Text Compression. *Computers & Security*, No. 7, 1988.

Witten, I.H., Neal, R.M. & Cleary, J.G.; Arithmetic Coding for Data Compression. *Communications of the ACM*, Vol. 30, No. 6, June 1987, p.520-540.

Appendix A

GLOSSARY

asymptotically optimal code. A code is *asymptotically optimal* when $H \to 1$ as the length approaches infinity for a given probability distribution.

a priori. Before an event.

a posteriori. After an event.

Blending. In an adaptive model, as the size of the sample increases, it becomes more meaningful to use a higher order model. When the sample is small, it is better to use a lower context. Blending is the combining of these two contexts.

Blending strategies use a number of model of different order. Predictions can be combined by assigning a weight to each model and calculate the weighted sum of the probabilities. This important for escaping from a context. [Bell, Cleary & Witten]

Binary Symmetric Source (BSS). A source which outputs a sequence of statistically independent and symmetrically distributed random binary variables.

Channel Coding. The means by which the information is transferred from the sender to the receiver. For example: Trellis coding. (Note the difference to *source coding*).

Code Space. The estimated probability of the symbol .

Codeword. The output of a compression encoder of some form of data compression on the input symbol.

Conditioning Classes. Conditioning classes are a context for each symbol which consists of a collection of probability distributions. They condition the probability estimates. [Bell, Cleary & Witten].

DES. Data Encryption Standard.

Deterministic Processes. A deterministic process can be predicted on the basis of past data.

Discrete Channel. All input and output symbols are part of a finite alphabet of symbols.

Discrete Memoryless Channel. A discrete channel in which each output symbol is statistically dependant only on the corresponding symbol of the input sequence.

Entropy. Measurement of the information content in a message. The higher the entropy, the more information it contains.

Enumerative Codes. The text is compressed by enumerating (listing) the entire set of possible strings. [Bell, Cleary & Witten] and [Rissanen 1981].

Ergodic Processes. An ergodic process has the property that a random process (ensemble) observed for a fixed time is identical to observing one sample function all the time. A process cannot be ergodic unless it is stationary. A stationary process is not necessarily ergodic.

Escape Contexts. Escape probability is the probability that a previously unseen symbol will occur (manifestation of the zero-frequency problem). This governs whether the system escapes to a smaller context to determine its probabilities.

Homophonic substitution. A one to many mapping of a symbol thus expanding the possible alphabet.

Keyspace. A finite set of possible keys.

LRU. Least Recently Used.

Markov Processes. A Markov process depends only on the preceding discrete sample.

Noiseless Source Coding Theorem. This theorem states that a message cannot be encoded so that the average length is less than the entropy of the message.

Noiseless Channel. The reception of data over the channel is identical to that transmitted.

Order of the Model. The order refers to how many previous symbols are taken into account. An order-0 ignores all previous symbols. With an order-1 (di-gram) model the previous symbol is used, for example, if the symbol was a 'q', the probability that the next symbol would be a 'u' is high.

Phrase. A string of symbols of arbitrary length.

Prefix free codes. A code used in the compressed representation may not be a prefix of any other code. The code can then be uniquely decoded. They are also known as instantaneous codes [Welch]. However, an instantaneous code is not necessary a prefix code.

Source Coding. In terms of information theory, a source deals with a stream of symbol from a finite alphabet. The source has some random mechanism based on the statistics of the event being modelled.

Stationary Processes. A process is stationary when all the probability densities of a random process are time independent.

Symbol. An atomic unit of information, e.g. a character, a pixel, number, etc.

Token. In terms of dictionary based compression, it is an object used to decode a phrase derived from the dictionary.

Trie. Derived from the word reTRIEval. A trie is a multiway digital search tree with a path from a root to a node for *each* string in the tree. This means that there is no single root, but multiple roots.

Zero-frequency Problem. If a symbol has never been observed (by an adaptive model) it has a zero probability and since symbols are encoded in $-\log p$ bits, the code length approaches infinity.

Appendix B

ARITHMETIC CODING CALCULATIONS

The following data is taken from the example in [Penzhorn, p.340]. These calculations show the encoding and decoding of a message using the standard coding step as given on Page 28. All the calculations are done with floating point arithmetic for easy calculation. The final example uses a message encoded with homophones and uses the original symbol probabilities to try to decode the message.

Table 4. Arithmetic coding calculations - symbol probabilities.

Source Symbols					**Homophones**		
Symbol	$P(u_i)$ **(decimal)**	$P(u_i)$ **(binary)**	$P(u_i)$ **(truncated)**	**Cum. Prob.**	**Symbol**	$P(v_{ij})$	**Cum. Prob.**
u_1	0.3333	0.01010101	0.0101	0.0	$v_{11}=\frac{1}{4}$	0.01	0.0
					$v_{12}=\frac{1}{16}$	0.0001	0.25
u_2	0.2	0.00110011	0.0011	0.3333	$v_{21}=\frac{1}{8}$	0.001	0.3125
					$v_{22}=\frac{1}{16}$	0.0001	0.4375
u_3	0.1333	0.00100010	0.0010	0.5333	$v_{31}=\frac{1}{8}$	0.001	0.5
u_4	0.2	0.00110011	0.0011	0.6666	$v_{41}=\frac{1}{8}$	0.001	0.625
					$v_{42}=\frac{1}{16}$	0.0001	0.75
u_5	0.1333	0.00100010	0.0010	0.8666	$v_{51}=\frac{1}{8}$	0.001	0.8125
							0.9375

Consider these messages which will be encoded from the statistics in the previous table:

1. The original message: $u_1 u_1 u_2 u_3 u_3 u_5 u_2$ is encoded as follows:

Symbol	Low	High	Range
u_1	0.0	0.3333	1.0
u_1	0.0	0.111089	0.3333
u_2	0.0370259	0.0592437	0.111089
u_3	0.0488747	0.0518363	0.0222178
u_3	0.0504541	0.0508489	0.00296163
u_5	0.507962	0.0508489	0.000394784
u_2	0.0508138	0.0508243	5.26644e-5

Thus, the message $u_1 u_1 u_2 u_3 u_3 u_5 u_2$ can be represented by 0.050814 which is within the final interval.

2. Homophonic coding the original message with $v_{12}\, v_{11}\, v_{21}\, v_{31}\, v_{31}\, v_{51}\, v_{22}$ respectively. The encoding process is given in the following table.

Symbol	Low	High	Range
v_{12}	0.25	0.3125	1.0
v_{11}	0.25	0.265625	0.0625
v_{21}	0.254883	0.256836	0.015625
v_{31}	0.255859	0.256104	0.00195312
v_{31}	0.255981	0.256012	0.000244141
v_{51}	0.256006	0.25601	3.05176e-5
v_{22}	0.256008	0.256008	3.8147e-6

The message, $v_{12}\, v_{11}\, v_{21}\, v_{31}\, v_{31}\, v_{51}\, v_{22}$ can be represented by 0.256008. Notice how small the *range* becomes and how close together the *low* and *high* become. This is expected as we have coded symbols with much smaller values than the original message. If more symbols were to be encoded, the range would be reduced even further and require more digits to be represented.

3. Homophonic decoding the message $v_{12}\, v_{11}\, v_{21}\, v_{31}\, v_{31}\, v_{51}\, v_{22}$, but using the original message for the cumulative probabilities. The decoding process is given in the following table.

Input Value	Symbol	Cumulative Probability High	Cumulative Probability Low	New input value	Range
0.256008	u_1	0.3333	0.0	0.768101	0.3333
0.768101	u_4	0.8666	0.6666	0.507504	0.2
0.507504	u_2				

These results show that the decoder model must know all the homophones for correct decoding. Only the first symbol will be decoded correctly. The symbol u_4 is decoded incorrectly (we expected to receive u_1 again) and thus the remainder of the message will also be incorrect.

Appendix C

SOURCE CODE

The source code for the implementation of universal dyadic homophonic coding comprises are three modules. The encoder module, the standard model and the homophonic model for generating the homophones. Due to the unsatisfactory results of the encoding process, the decoder source code is not included in this dissertation.

Encoder

```
/************************************************************************
         HOMOPHONIC ARITHMETIC ENCODER

   Arithmetic encoder implementation of Bell, Cleary & Witten.

File:   hce.cpp
Author: C.C. Stevens

************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include "modelhc.h"


/* DECLARATIONS USED FOR ARITHMETIC ENCODING AND DECODING */

#define Top_value (((long)1<<Code_value_bits)-1)      /* Largest code value */


/* HALF AND QUARTER POINTS IN THE CODE VALUE RANGE. */

#define First_qtr (Top_value/4+1)       /* Point after first quarter        */
#define Half      (2*First_qtr)         /* Point after first half           */
#define Third_qtr (3*First_qtr)         /* Point after third quarter        */


void start_encoding(void);
void encode_symbol(int symbol, int cum_freq[]);
void done_encoding(void);
void done_outputing_bits(void);

long    wInputByteCounter;

static  int fUseHomophonicModel;

/* BIT OUTPUT ROUTINES. */

/* THE BIT BUFFER. */

int buffer;                     /* Bits buffered for output             */
int bits_to_go;                 /* Number of bits free in buffer        */


/* INITIALIZE FOR BIT OUTPUT. */
```

```
void start_outputing_bits(void)
{
    buffer      = 0;                               /* Buffer is empty to start */
    bits_to_go  = 8;                               /* with.                    */
}

/* OUTPUT A BIT. */

inline void output_bit(int bit)
{
    buffer >>= 1;
    if (bit)
        buffer |= 0x80;                            /* Put bit in top of buffer.*/

    bits_to_go--;
    if (bits_to_go == 0)
    {                                              /* Output buffer if it is   */
        putc((char) buffer, stdout);               /* now full.                */
        bits_to_go = 8;
    }
}

/* FLUSH OUT THE LAST BITS. */

void done_outputing_bits(void)
{
    putc((char) (buffer >> bits_to_go), stdout);
}

/* CURRENT STATE OF THE ENCODING. */

code_value low, high;           /* Ends of the current code region         */
long bits_to_follow;            /* Number of opposite bits to output after */
                                /* the next bit.                           */
/* OUTPUT BITS PLUS FOLLOWING OPPOSITE BITS. */

void bit_plus_follow(int bit)
{
    output_bit(bit);                               /* Output the bit.          */
    while (bits_to_follow > 0)
    {
        output_bit(!bit);                          /* Output bits_to_follow    */
        bits_to_follow -= 1;                       /* opposite bits. Set       */
    }                                              /* bits_to_follow to zero.  */
}

int main(int argc, char *argv[])
{
    int ch, symbol;

    wInputByteCounter = 0;

    fprintf(stderr, "Arithmetic coding on ");
    if (argc > 1)
    {
        freopen(argv[1], "rb", stdin);
        fprintf(stderr, "%s", argv[1]);
    }
    else
        fprintf(stderr, "stdin");

    fprintf(stderr, " to ");

    if (argc > 2)
    {
        freopen(argv[2], "wb", stdout);
        fprintf(stderr, "%s", argv[2]);
    }
    else
        fprintf(stderr, "stdout");
    fprintf(stderr, "\n");

    InitModelHC();                              /* Init. homophonic model    */
    fUseHomophonicModel = FALSE;

    start_model();                              /* Set up other modules.    */
    start_outputing_bits();
    start_encoding();
    for (;;)
    {                                           /* Loop through characters. */
        ch = getc(stdin);                       /* Read the next character. */
        if (ch == EOF)
            break;                              /* Exit loop on end-of-file.*/

            wInputByteCounter++;

        symbol = char_to_index[ch];             /* xlate to an index        */
        if (!fUseHomophonicModel)
            encode_symbol(symbol, cum_freq);    /* Encode that symbol.      */
        else
            EncodeCharacter(ch);

        update_model(symbol);               /* Update the model.        */
        fUseHomophonicModel = CreateHomophonicModel();
    }
    if (!fUseHomophonicModel)
        encode_symbol(EOF_symbol, cum_freq);    /* Encode the EOF symbol.   */
    else
        encode_symbol(wEOF, wCumFreqHC);        /* Encode the EOF symbol.   */

    done_encoding();                            /* Send the last few bits.  */
    done_outputing_bits();
    putc('\n', stderr);
    return 0;
}

/* ARITHMETIC ENCODING ALGORITHM. */

/* START ENCODING A STREAM OF SYMBOLS. */

void start_encoding(void)
{
    low             = 0;                        /* Full code range.         */
    high            = Top_value;
    bits_to_follow  = 0;                        /* No bits to follow next.  */
}


/* ENCODE A SYMBOL. */

void encode_symbol(int symbol, int cumfreq[])
{
    unsigned long   range;              /* Size of the current code region  */
    unsigned int count = 0xffff;

    range   = (long)(high - low) + 1;

    /* Narrow the code region to that allotted to this symbol.             */
    high    = low + (range * cumfreq[symbol-1]) / cumfreq[0]-1;
    low     = low + (range * cumfreq[symbol]) / cumfreq[0];

    for (;;)
    {                                           /* Loop to output bits.     */
        if (high < Half)
        {
            bit_plus_follow(0);                 /* Output 0 if in low half. */
        }
        else if (low >= Half)
        {                                     /* Output 1 if in high half.*/
            bit_plus_follow(1);
            low  -= Half;
            high -= Half;                       /* Subtract offset to top.  */
        }
        else if (low >= First_qtr                 /* Output an opposite bit   */
              && high<Third_qtr)
        {                                         /* later if in middle half. */
            bits_to_follow += 1;
            low  -= First_qtr;                    /* Subtract offset to middle*/
            high -= First_qtr;
        }
        else break;                               /* Otherwise exit loop.     */

        low  = 2*low;
        high = 2*high+1;                          /* Scale up code range.     */
        count --;
        if (count == 0)
        {
            fprintf(stderr, "Hanging, count = %d",wInputByteCounter);
            exit (-1);}
        }
}


/* FINISH ENCODING THE STREAM. */

void done_encoding(void)
{
    bits_to_follow += 1;                          /* Output two bits that     */
    if (low < First_qtr)
        bit_plus_follow(0);                       /* select the quarter that  */
    else
        bit_plus_follow(1);                       /* the current code range   */
}                                                 /* contains.                */
```

Order-0 Model

```
/*************************************************************************
         HOMOPHONIC BASED ARITHMETIC ENCODER

         ORDER-0 ADAPTIVE MODEL

Based on the arithmetic encoder implementation of Bell, Cleary & Witten.

File:   modelhc.h
Author: C.C. Stevens

*************************************************************************/
#define FALSE                0
#define TRUE                 1
#define RefreshModel         10
#define HC_BitSize           4096 //8192

/* SIZE OF ARITHMETIC CODE VALUES. */

#define Code_value_bits      16          /* Number of bits in a code value   */
typedef unsigned long code_value;        /* Type of an arithmetic code value */

#define NumOfHomophones      16          /* Number of homophones per char    */
#define No_of_chars          256         /* Number of character symbols      */
#define EOF_symbol           (No_of_chars+1) /* EOF symbol    */

#define No_of_symbols        (No_of_chars+1)   /* Total number of symbols    */

#define Max_frequency        16383       /* Maximum allowed frequency count  */

void    start_model(void);
void    update_model(int symbol);

/* TRANSLATION TABLES BETWEEN CHARACTERS AND SYMBOL INDEXES. */

extern  int     char_to_index[No_of_chars];
extern  int     index_to_char[No_of_symbols + 1];
extern  int     cum_freq[No_of_symbols + 1];
extern  int     freq[No_of_symbols + 1];

extern  int     wCumFreqHC[No_of_symbols * NumOfHomophones + NumOfHomophones];
extern  int     wEOF;

extern  void    InitModelHC(void);
extern  int     CreateHomophonicModel(void);
extern  void    EncodeCharacter(int ch);

/*************************************************************************
         HOMOPHONIC BASED ARITHMETIC ENCODER

         ORDER-0 ADAPTIVE MODEL

   Arithmetic encoder implementation of Bell, Cleary & Witten.

File:   model.cpp
Author: C.C. Stevens

*************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include "modelhc.h"

/* TRANSLATION TABLES BETWEEN CHARACTERS AND SYMBOL INDEXES. */

/* To index from character          */
int char_to_index[No_of_chars];

/* To character from index    */
int index_to_char[No_of_symbols + 1];

/* ADAPTIVE SOURCE MODEL */

/* Symbol frequencies                       */
int freq[No_of_symbols + 1];

/* Cumulative symbol frequencies            */
int cum_freq[No_of_symbols + 1];

/* THE ADAPTIVE SOURCE MODEL */

/* INITIALIZE THE MODEL. */

void start_model(void)
{
    int i;

    /* Set up tables that translate between symbol                             */
    for (i = 0; i < No_of_chars; i++)
    {
        char_to_index[i]    = i+1;
        index_to_char[i+1]  = i;
    }

    /* Set up initial frequency counts to be one for all                       */
    for (i = 0; i <= No_of_symbols; i++)
    {
        freq[i]         = 1;
        cum_freq[i]     = No_of_symbols - i;
    }

    /* Freq[0] must not be the same as freq[NEXT SYMBOL].                      */
    freq[0] = 0;
}

/* UPDATE THE MODEL TO ACCOUNT FOR A NEW SYMBOL. */

void update_model(int symbol)
{
    int      i;                             /* New index for symbol      */

    if (cum_freq[0] >= Max_frequency)
    {                                       /* See if frequency counts   */
        int cum;                            /* are at their maximum.     */

        cum = 0;
        /* If so, halve all the counts (keeping them non-zero).          */
        for (i = No_of_symbols; i >= 0; i--)
        {
            freq[i]         = (freq[i]+1)/2;
            cum_freq[i]     = cum;
            cum            += freq[i];
        }
    }

    /* Find symbol's new index.                                          */
    for (i = symbol; freq[i] == freq[i-1]; i--)
        ;

    if (i < symbol)
    {
        int ch_i, ch_symbol;

        /* Update the translation tables if the symbol has moved    */
        ch_i                    = index_to_char[i];
        ch_symbol               = index_to_char[symbol];
        index_to_char[i]        = ch_symbol;
        index_to_char[symbol]   = ch_i;
        char_to_index[ch_i]     = symbol;
        char_to_index[ch_symbol]= i;
    }

    freq[i]++;                          /* Increment the frequency   */
    while (i > 0)
    {                                   /* count for the symbol and */
        i   = i-1;                      /* update the cumulative    */
        cum_freq[i]++;                  /* frequencies.             */
    }
}
```

Homophonic Model

```
/*************************************************************************
         HOMOPHONIC BASED ARITHMETIC ENCODER

         ORDER-0 HOMPHONIC MODEL

Based on the arithmetic encoder implementation of Bell, Cleary & Witten.

File:   hcmodel.cpp
Author: C.C. Stevens

*************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "modelhc.h"

extern  void    encode_symbol(int symbol, int cum_freq[]);
extern  long    wInputByteCounter;


int wCumFreqHC[No_of_symbols * NumOfHomophones + NumOfHomophones];
        unsigned int wFreqHC[No_of_symbols + NumOfHomophones];
static  unsigned int wHomophones[No_of_symbols][NumOfHomophones];

int     wIndexToChar[No_of_symbols * NumOfHomophones + NumOfHomophones];
int     wCharToIndex[No_of_chars * NumOfHomophones];
```

```
static  unsigned int    wFreqSort[No_of_symbols * NumOfHomophones + NumOfHomophones];
static  unsigned int    wCharCounter;
static  unsigned int    wLargestFreq;
static  int     fUseHomophonicModel;
        int     wHomophonesPerSymbol[No_of_symbols];
        int     wEOF;

static  void    ScaleToBinary(void);
static  void    GenerateHomophones(int symbol);

int debug;

void    InitModelHC(void)
{
    int i, j;

    randomize();                            /* start the insecure RNG           */
    wCharCounter        = 0;
    fUseHomophonicModel = FALSE;

    for (i = 0; i < No_of_symbols; i++)
    {
        wHomophonesPerSymbol[i] = 1;    /* always at least one homophone    */
        wFreqHC[i]              = 0;
        for (j = 0; j < NumOfHomophones; j++)
        {
            wHomophones[i][j]   = 0;
        }
    }

    for (i = 0; i < (No_of_symbols * NumOfHomophones + NumOfHomophones); i++)
    {
        wCumFreqHC[i]   = 0;
    }
}

int CreateHomophonicModel(void)
{
    int             i, j, k;
    unsigned int    cum;

    wCharCounter++;
    if (wCharCounter > RefreshModel)        /* time to generate homophones  */
    {
        wCharCounter    = 0;
        if (fUseHomophonicModel == FALSE)
            fUseHomophonicModel = TRUE;     /* from now on, use homophones  */

        ScaleToBinary();                    /* convert to binary powers     */

        for (i = 1; i < (No_of_symbols); i++)
        {
            wIndexToChar[i] = index_to_char[i];
            wCharToIndex[i] = char_to_index[i];
            GenerateHomophones(i);
        }
        wCharToIndex[0]= char_to_index[0];

        /* Calculate a new set of cumulative frequencies for all homophones */
        cum = 0;
        for (i = 1, k = 1; i < No_of_symbols; i++)
        {
            for (j = wHomophonesPerSymbol[i]-1; j >= 0; j--)
            {
                wFreqSort[k++] = wHomophones[i][j];
            }
        }
        if (k<5)
            j=k;

        k--;
        wEOF    = k;
        wCumFreqHC[wEOF] = 0;

        for (; k > 0; k--)
        {
            cum = cum + wFreqSort[k];
            wCumFreqHC[k-1] = cum;
        }
        if (cum >= 0x8000)
        {   fprintf(stderr, "\nCum freq overflow %d", cum);
            exit(-1);
        }
    }
    return (fUseHomophonicModel);
}

void    EncodeCharacter(int ch)
{
    int     symbol, i;

    symbol  = wCharToIndex[ch];
    i  = (symbol * wHomophonesPerSymbol[symbol]) + random(wHomophonesPerSymbol[symbol]);
    encode_symbol(i, wCumFreqHC);
}

/* Based on the cumulative frequency total - select a power of 2 value. This
   value is used to scale all the frequency counts so that they are all based
   on a power of 2 value.                                                   */
static  void    ScaleToBinary(void)
{
    double      scale;
    unsigned int i, cum;

    if (cum_freq[0] <= 512)
        wLargestFreq    = 512;
    else if (cum_freq[0] <= 1024)
        wLargestFreq    = 1024;
    else if (cum_freq[0] <= 2048)
        wLargestFreq    = 2048;
    else if (cum_freq[0] <= 4096)
        wLargestFreq    = 4096;
    else if (cum_freq[0] <= 8192)
        wLargestFreq    = 8192;
    else if (cum_freq[0] <= 16384)
        wLargestFreq    = 16384;
    else
    {
        wLargestFreq    = 16384;
        fprintf(stderr,"\nScaling error cum freq = %x", cum_freq[0]);
    }

    scale   = (double)wLargestFreq / (double)cum_freq[0];
    cum     = 0;

    for (i = 1; i < (No_of_symbols); i++)
    {
        wFreqHC[i]  = floor((double)freq[i] * scale);
        if (wFreqHC[i] == 0)
            wFreqHC[i] = 1;
        cum         = cum + wFreqHC[i];
    }
    if (cum >= 0x8000)
    {
        for (i = 1; i < (No_of_symbols); i++)
        {
            wFreqHC[i]  = (wFreqHC[i] + 1) / 2;
        }
    }
}


/* Generate homophones use binary powers                              */
```

```
static  void    GenerateHomophones(int symbol)
{
    unsigned int    i, count, homophone;

    homophone    = wLargestFreq / 2;      /* init to largest power of 2 value */
    count        = wFreqHC[symbol];

    for (i=0; i < NumOfHomophones; i++)
        wHomophones[symbol][i] = 0;       /*debug clearing */

    i            = 0;
    while (count)                         /* reduce count to powers of 2      */
    {
        while (homophone > count)
        {
            homophone >>= 1;              /* get new possible homophone       */
        }
        wHomophones[symbol][i++]     = homophone;
        if (i > NumOfHomophones)
        {
            fprintf(stderr, "\nToo many homophones for memory alloced");
        }
        count -= homophone;               /* remove selected homophone and    */
    }                                     /* search for the next homophone    */
    wHomophonesPerSymbol[symbol]     = i;
    if (i == 0)
     fprintf(stderr, "\nNo Homophones!");
}
```