Universal coding with different modelers in data compression
by Reggie ChingPing Kwan

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science
Montana State University

Abstract:
In data compression systems, most existing codes have integer code length because of the nature of block codes. One of the ways to improve compression is to use codes with noninteger length. We developed an arithmetic universal coding scheme to achieve the entropy of an information source with the given probabilities from the modeler. We show how the universal coder can be used for both the probabilistic and nonprobabilistic approaches in data compression. to avoid a model file being too large, nonadaptive models are transformed into adaptive models simply by keeping the appropriate counts of symbols or strings. The idea of the universal coder is from the Elias code so it is quite close to the arithmetic code suggested by Rissanen and Langdon. The major difference is the way that we handle the precision problem and the carry-over problem. Ten to twenty percent extra compression can be expected as a result of using the universal coder. The process of adaptive modeling, however, may be forty times slower unless parallel algorithms are used to speed up the probability calculating process.

UNIVERSAL CODING WITH DIFFERENT MODELERS IN DATA COMPRESSION

by

Reggie ChingPing Kwan

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

July 1987

APPROVAL

of a thesis submitted by

Reggie ChingPing Kwan

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citation, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

August 3rd 1987
Date

J. Dunbigh Stanley
Chairperson, Graduate Committee

Approved for the Major Department

August 3rd 1987
Date

J. Dunbigh Stanley
Head, Major Department

Approved for the College of Graduate Studies

12 August 1987
Date

M. S. Malone
Graduate Dean

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from the thesis are allowable without special permission, provided that accurate acknowledgment of source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his/her absence, by the Director of Libraries when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature Reed Ryan

Date July 30, 87

TABLE OF CONTENTS

TABLE OF CONTENTS--Continued

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

In data compression systems, most existing codes have integer code length because of the nature of block codes. One of the ways to improve compression is to use codes with noninteger length. We developed an arithmetic universal coding scheme to achieve the entropy of an information source with the given probabilities from the modeler. We show how the universal coder can be used for both the probabilistic and nonprobabilistic approaches in data compression. To avoid a model file being too large, nonadaptive models are transformed into adaptive models simply by keeping the appropriate counts of symbols or strings. The idea of the universal coder is from the Elias code so it is quite close to the arithmetic code suggested by Rissanen and Langdon. The major difference is the way that we handle the precision problem and the carry-over problem. Ten to twenty percent extra compression can be expected as a result of using the universal coder. The process of adaptive modeling, however, may be forty times slower unless parallel algorithms are used to speed up the probability calculating process.

Chapter 1

INTRODUCTION TO DATA COMPRESSION

A Basic Communication System

Communications are used to provide terminal users access to computer systems that may be remotely located, as well as to provide a means for computers to communicate with other computers for the purpose of load sharing and accessing the data and/or programs stored at a distant computer. The conventional communication system is modeled by:

1. An information source
2. An encoding of this source
3. A channel over, or through, which the information is sent
4. A noise (error) source that is added to the signal in the channel
5. A decoding and hopefully a recovery of the original information from the contaminated received signal
6. A sink for the information

This model, shown in figure 1, is a simplified version of any communication systems. In (3), we are particularly interested with sending information either from now to then (storage), or from here to there (transmission).

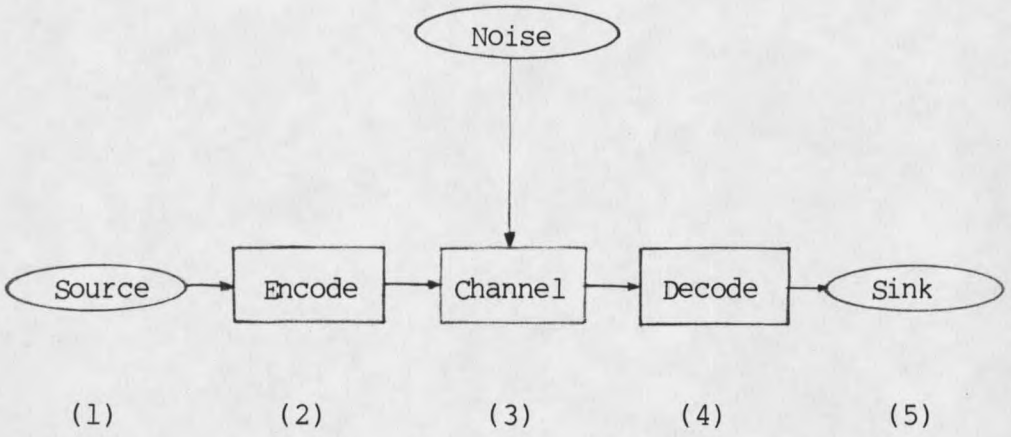


Figure 1. A basic communication system.

The Need For Data Compression

There are three major reasons that we bother to code:

1. Error detection and correction
2. Encryption
3. Data compression

To recover from noise, parity check and Hamming (4,7) codes [1] are introduced to detect and correct errors respectively. To avoid unauthorized receivers, encryption schemes like public key encryption system [2] are used. Finally, to reduce the data storage requirements and/or the data communication costs, there is a need to reduce the size of voluminous amounts of data. Thus, codes like Shannon-Fanno [3], Huffman [3], LZW [4], and Arithmetic [5] and [6] etc. are employed.

Data compression techniques are most useful for large archival files, I/O bound systems with spare CPU cycles, and transfer of large files over long distance communication links especially when low band width is available.

This paper discusses several data compression systems and develops a universal coder for diffferent modelers.

Data Compression and Compaction - A definition

Data compression/compaction can be defined as the process of creating a condensed version of the source data by reducing redundancies in the data. In general, data compression techniques can be divided into two categories -- nonreversible and reversible. In nonreversible techniques (usually called data compaction), the size of the physical representation of data is reduced while a subset of "relevant" information is preserved, e.g. CCC algorithm [7], and Rear-compaction [8]. In reversible techniques, all information is considered relevant, and compression/decompression recovers the original data, e.g. LZW algorithm [4].

Theoretical Basis

The encoding of an information source can be illustrated by figure 2.

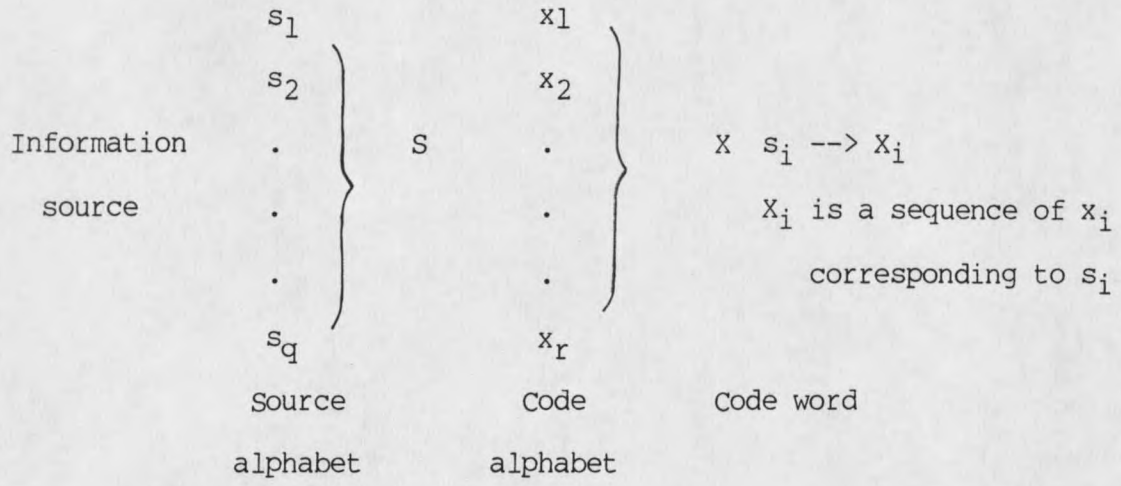


Figure 2. Coding of an information source.

In fact, figure 2 is the component labeled (2) in figure 1. The simplest and most important coding philosophy in data compression is to assign short code words to those events with high probabilities and long code words to those events with low probabilities [3]. An event, however, can be a value, source symbol or a group of source symbols to be encoded.

Tables 1 and 2 are used to illustrate the idea to achieve data compression when the probabilities are known. Consider the language Kikian of a very primitive tribe Kiki. The Kikian can only communicate in four words # = kill, @ = eat, z = sleep, and & = think. The source alphabet S = { #, @, z, & }. Assume an anthropologist studies Kikian and he encodes their daily activities with a binary code. Then, the code alphabet X = { 0, 1 }. Without knowing the

probabilities of the tribe's activities, or source symbols, we constructed the code for the source symbols as table 1.

With the probabilities of each activity, or symbol, being known, we can employ our coding philosophy to construct a new code, as shown in table 2.

Table 1. Kikian code.

Symbol	Code A
#	00
@	01
z	10
&	11

Table 2. Compressed Kikian code.

Symbol	Probability	Code B
#	0.125	110
@	0.25	10
z	0.5	0
&	0.125	111

The average length L of a code word using any code is defined as

$$L = \sum_{i=1}^{q} |C(S_i)| * p(S_i) \qquad (1.1)$$

where q = number of source symbols

S_i = source symbols

$C(S_i)$ = code for S_i

$|C(S_i)|$ = length of the code for symbol S_i

$p(S_i)$ = probability of symbol S_i

The average length L_A of a code word using code A is obviously 2 bits because $|C(S_i)| = 2$ for all i and $p(S_i)$ is always one. On the other hand, the average length L_B of a code word using code B is 1.75 bits, calculate as

$$L_B = 0.125 * 3 + 0.25 * 2 + 0.5 * 1 + 0.125 * 3.$$

That is, for our encoding system for the Kikian we have found a method using an average of only 1.75 bits per symbol, rather than 2 bits per symbol. Both codes, however, are instantaneous (see Appendix B), and therefore uniquely decodable. It is because all code words from both codes are formed by the leaves of their code tree as in figure 3.

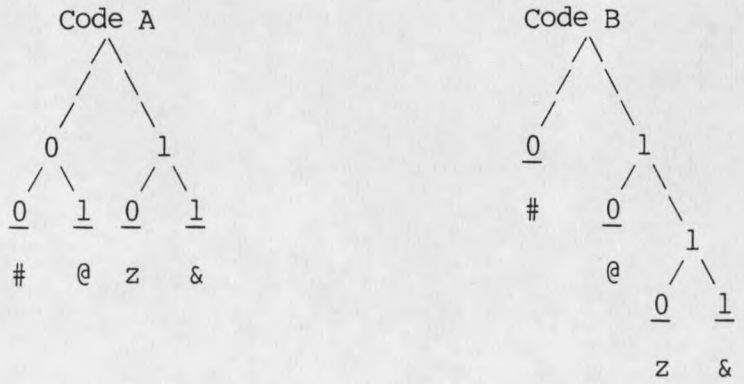


Figure 3. Code trees of Code A and Code B.

A typical day of the Kikian could be sleep, eat, sleep, kill, sleep, think, eat, and sleep. If a Kikian has a diary, z@z#z&@z will be the symbols appear on a typical day. On the anthropologist's note book, it would be 1001100010110110 without considering the probabilities of different activities. With the probabilities being known, however, the daily activities of the Kikian can be represented by 01001100111100. In other words, to represent the same information, the daily activities of the Kikian, we can either use 16 bits or 14 bits.

Code B, in this case, is clearly better than code A even though they are both uniquely decodable which is the major criteria in data compression. Nevertheless, at this point we cannot conclude if code B gives us the best compression for the given probabilities, though it turns out to be true.

To determine what is the best compression we can get for the given probabilities, let us look at the information content of each event first. Let $p(S_i)$ be the probability of the symbol S_i. We can say that the information content of S_i is:

$$I(S_i) = - \log p(S_i) \qquad (1.2)$$

bits of information if log base two is used [3].

Code B uses only one bit to represent z and three bits to represent @ because the information content of z is less than @.

$$I(z) = -\log_2 p(z) = -\log_2 0.5 = 1 \text{ bit}$$

$$I(@) = -\log_2 p(@) = -\log_2 0.125 = 3 \text{ bits}$$

Intuitively, if an event is highly unlikely to happen and happens, it deserves more attention or it contains more information.

To determine if a code has the shortest average code length L, we have to compare L with the entropy H(S). If symbol s_i occurs, we obtain an amount of information equal to

$$I(S_i) = - \log_2 p(S_i) \text{ bits}$$

The probabilities that S_i is $p(S_i)$, so the average amount of information obtain per symbol from the source is

$$\sum_S p(S_i)\ I(S_i) \quad \text{bits}$$

The quantity, the average amount of information per source symbol, is called entropy H(S) of the source

$$H(S) = - \sum_{i=1}^{q} p(S_i) \log_2 p(S_i) \quad \text{bits} \qquad (1.3)$$

In code B, $L_B = H(S) = 1.75$ bits, and therefore code B provides the best compression with the given probabilities according to Shannon First Theorem

$$H_r(S) =< L < H_r(S) + 1. \qquad (1.4)$$

The proof of the above theorm can be found in any information or coding book. It is obvious that the lower bound of L is the entropy. The upper bound of L, on the other hand, can be explained by the nature of block coded. The length of any block code is always an integer, even though the information content is not necessarily an integer because $|C(S_i i)|$ form (1.1) is calculated as $|C(S_i)| = |\log p(S_i)|$.

Components of a Data Compression System

The fundamental components of a data compression system are:

1. Information source
2. Modeler
3. Encoder
4. Compressed file
5. Decoder.

In a digital image compression system, a predictor may be used with the modeler to capture hopefully not so obvious redundancies. For the purpose of our discussion, we would concentrate on modelers, encoders and decoders. Components of a data compression system are shown in figure 4.

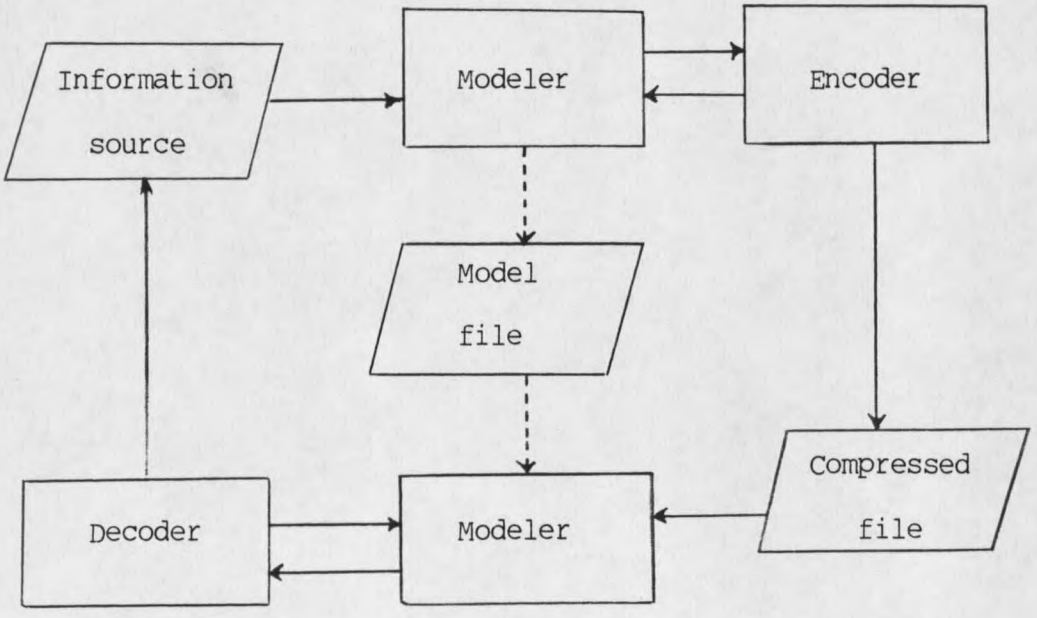


Figure 4. Components of a data compression system.

Information Source

For communication purposes, an information source can be thought of a random process {S(i)} that emits outcomes of random variables S(i) in a never ending stream. The time index i then runs through all the integers. In addition to the random variables S(i), the process also defined all the finite joint variables (S(i), S(j), ...). The most frequently studied information sources are either independent, or Markovian.

A source is said to be independent if $p(S_i) = p(S_i|S_j)$, where j = 1,...,q. Therefore, it is often called zero-memory source.

A more general type of information source with q symbols is one in which the occurrence of a source symbol S_i may depend on a finite number m of preceding symbols. Such a source (called Markovian, or an m^{th}-order Markov source) is specified by giving the source alphabet S and the set of conditional probabilities

$$p(S_i|S_{j1},S_{j2},\ldots,S_{jm}) \quad \text{for } i=1,2,\ldots,q;\ j_p=1,2,\ldots,q \qquad (1.5)$$

For an m^{th}-order Markov source, the probability of a symbol is known if we know the m preceding symbols.

Modeler

The modeler is responsible for capturing the "designated" redundancies of the information source. Designated redundancies can be viewed as the frequency measures used in the coding process.

In a straight forward data compression system like Huffman code [3], the modeler simply counts the occurrence of each symbol S_i, for

i=1,2,...,q. In other more sophisticated systems, e.g. LZW algorithm [4], the modeler is responsible for putting new strings in the string table. Though the algorithm does not attempt to subdivide the process into modeling and coding, modeling and coding are actually done separately and will be shown in chapter 3.

Depending on how the probabilities are calculated, we can distinguish between two cases: the adaptive and the nonadaptive models discussed in detail in chapter 3. In a nonadaptive model, a model file is generated as in figure 4. The model file contains the necessary probabilities for both the encoder and the decoder.

Encoder

The encoder encodes the symbols or strings with their probabilities given by the modeler. The basic data compression philosophy, symbols with higher probabilities are assigned with the shorter code words, applies here. The encoder also ensures the codes to be uniquely decodable for reversible compression.

In our data compression system, all the encoder needs from the modeler are the probabilities of all symbols and the previous symbols. The whole encoding process will be **adding** and **shifting.**

Compressed File

The compressed file can be thought of a file with a sequence of code words or code alphabets. With X the set of the code alphabets, the compressed file x should be simply a subset of X^*.

Decoder

The decoding process also involves the modeler regardless of the nature of the modeler (adaptive or nonadaptive). If nonadaptive modeler is used, a model file is required with the compressed file to provide all the necessary statistics.

In our data compression system, the decoder goes through a **subtract** and **shift** process to recover the original source.

Chapter 2

UNIVERSAL CODING

Coding Questions and Efficiency

The three major questions in coding theory are:

1. How good are the best codes?
2. How can we design "good" codes?
3. How can we decode such codes?

In data compression systems, the first question has been answered by Shannon's first theorem described in the previous chapter. Shannon's first theorem shows that there exists a common yardstick with which we may measure any information source. The value of a symbol (or string in some cases) from an information source S may be measured in terms of an equivalent number of r-ary digits needed to represent one symbol/string from the source; the theorem says that the average value of a symbol/string from S is H(S).

Let the average length of a uniquely decodable r-ary code for the source S be L. By Shannon's first theorem, L cannot be less than $H_r(S)$. Accordingly, we define E, the **Efficiency** of the code, by

$$E = H_r(S) / L \qquad (2.1)$$

We are trying to answer the other two questions in this thesis. In order to answer them, we need to study several existing data

compression systems because there are no absolute answers for the two questions. Before we do that, we shall look at what do we mean by "good" code in the next section.

Criteria of a Good Data Compression System

A good data compression system has to have the following three qualities:

1. the codes it produces have to be decodable.
2. the average code length should be as close to the entropy as possible, or E ~ 1.0.
3. the coding (encoding and decoding) process is done in "acceptable" time.

Decodability can easily be achieved by constructing all the code words by the leaves of the code tree in the case of block codes. We shall develop ways to ensure decodability in later sections.

Later in the chapter, we describe a universal coding technique that guarantees the average code length to be the same as the entropy for the given probabilities provided by the modeler. That is to say, the efficiency of the code equals one according to (2.1). Besides looking at the efficiency of codes, we can also compare their compression ratios defined by

$$PC = (|OF| - |CF|) / |OF| \qquad (2.2)$$

where PC = Percentage of compression,

OF = Original File, and

CF = Compressed File.

There is no absolute way to measure "acceptable" time. Some codes are relatively more acceptable than others, while it is clear that there is a trade-off between compression ratio and time. The longer the modeler analyzes the information source the smaller the compressed file is going to be.

Codes with Combined Modelers and Coders

A lot of data compression systems do not separate their processes into modeling and coding. In this chapter, we look at several popular data compression systems and then discuss the universal coding scheme that we use.

Huffman Code for Zero-memory Source

Huffman code [1] is one of the variable-length codes that enforces the data compression philosophy -- the most frequent symbols to have the shortest encodings. Consider the source S with the symbols $S_1, S_2, \ldots, S_q$ and symbol probabilities $p_1, p_2, \ldots, p_q$. Let the code X with the symbols $x_1, x_2, \ldots, x_r$. The generation of Huffman code can be divided into two processes, namely reduction and splitting. The reduction process involves:

a. Let the symbols be ordered that $p_1 >= p_2 >= \ldots >= p_q$.

b. Combine the last r symbols into one symbol.

c. Rearrange the remaining source containing q-r symbols in decreasing order.

d. Repeat (c) and (d) until there are less than or equal to r symbols.

After the reduction process, we then go backward to perform the splitting process:

a. List the r code symbols in a fixed order.

b. Assign the r symbols to the final reduced source.

c. go backward with the one of the symbols splits into r symbols.

d. Append the r code symbols to the last r source symbols.

e. Repeat (c) and (d) until there are q source symbols.

Figures 5 and 6 are the reduction and splitting processes of an example with $S = \{S_1, S_2, S_3, S_4, S_5\}$, $p_1 = 0.4$, $p_2 = 0.2$, $p_3 = 0.2$, $p_4 = 0.1$, $p_5 = 0.1$, and $X = \{0, 1\}$.

The Huffman code for S is formed where $C(S_1) = 1$, $C(S_2) = 01$, and so on. Since the lengths of the codes are not necessary the same, Huffman code is a perfect example of variable-length code. The average code length of the Huffman code:

$$L = 1(0.4) + 2(0.2) + 3(0.2) + 4(0.1) + 4(0.1)$$

$$= 2.5 \text{ bits/symbol.}$$

The entropy of the above source:

$$H(S) = (0.4)\log(1/0.4) + 2 * (0.2)\log(1/0.2) + 2 * (0.1)\log(1/0.1)$$

$$= 2.12 \text{ bits/symbols.}$$

The efficiency of the code:

$$E = 2.12 / 2.5$$

$$= 0.848$$

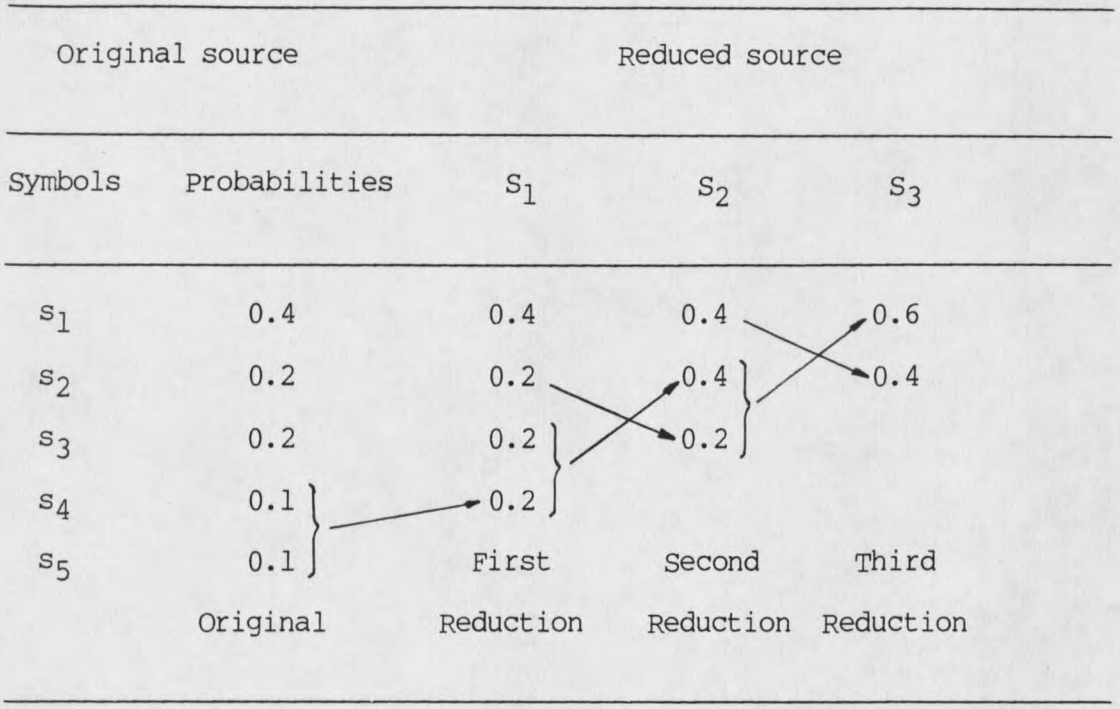


Figure 5. Reduction process.

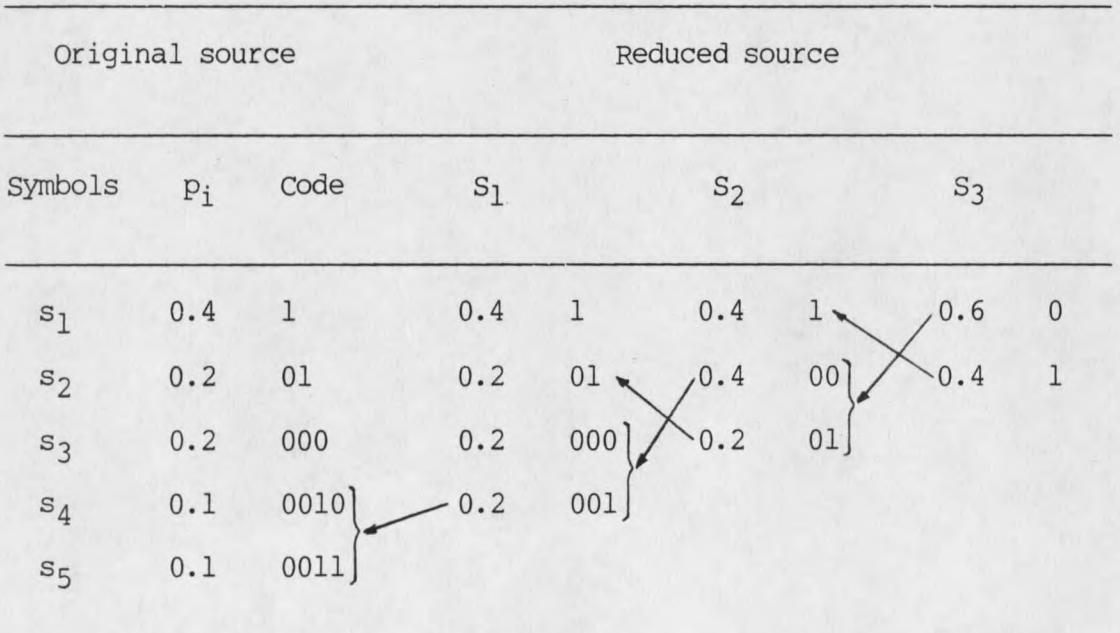


Figure 6. Splitting process.

Huffman Code for Markov Source

The best way to illustrate the behavior of a m^{th}-order Markov source defined in chapter 1 is through the use of state diagram. In a state diagram we represent each of the q^m possible states of the source by a single circle, and the possible transitions from state to state by arrows as in figure 7. As an example of a simple one-memory source, suppose we have an alphabet of three symbols, a, b, and c (S = {a, b, c}). Let the probabilities be:

p(a|a) = 1/3	p(b|a) = 1/3	p(c|a) = 1/3
p(a|b) = 1/4	p(b|b) = 1/2	p(c|b) = 1/4
p(a|c) = 1/4	p(b|c) = 1/4	p(c|c) = 1/2

By solving the following equations formed by the above probabilities,

$$p_a * p(a|a) + p_b * p(a|b) + p_c * p(a|c) = p_a$$

$$p_a * p(b|a) + p_b * p(b|b) + p_c * p(b|c) = p_b$$

$$p_a * p(c|a) + p_b * p(c|b) + p_c * p(c|c) = p_c$$

$$p_a + p_b + p_c = 1$$

we get

$$p_a = 3/11, \quad p_b = 4/11, \quad p_c = 4/11.$$

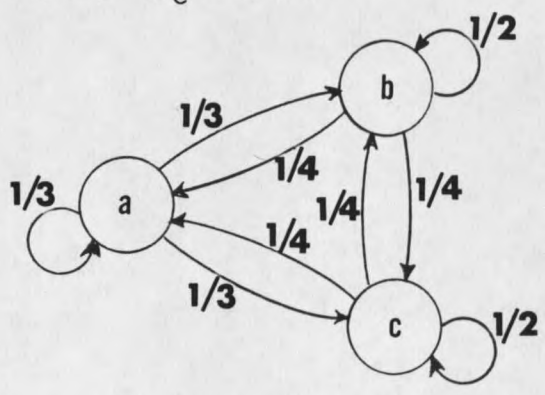


Figure 7. State diagram of a first-order Markov source.

There are, of course, three states (3^1) in this case, labeled a, b, and c and indicated by the circles. Each directed line is a transition from one state to another state, whose probability is indicated by the number alongside the line.

For each state in the Markov system we can use the appropriate Huffman code obtained from the corresponding transition probabilities for leaving that state. The generation of the codes is exactly the same as the zero-memory source at each state. By the same token, we can break down any order Markov source into q^m zero-memory states. For state a we get the binary encoding

a = 1 $p_a = 1/3$

b = 00 $p_b = 1/3$

c = 01 $p_c = 1/3$

By (1.1), $L_a = 1.67$. For state b,

a = 10 $p_a = 1/4$

b = 0 $p_b = 1/2$

c = 11 $p_c = 1/4$

By (1.1), $L_b = 1.5$. For state c,

a = 10 $p_a = 1/4$

b = 11 $p_b = 1/4$

c = 0 $p_c = 1/2$

By (1.1), $L_c = 1.5$. The average code length of the Huffman code for the above Markov source

$L_M = p_a * L_a + p_b * L_b + p_c * L_c = 1.55$ bits/symbols.

Assuming the starting state is a, the encoding process can be shown by another state diagram as in figure 8.

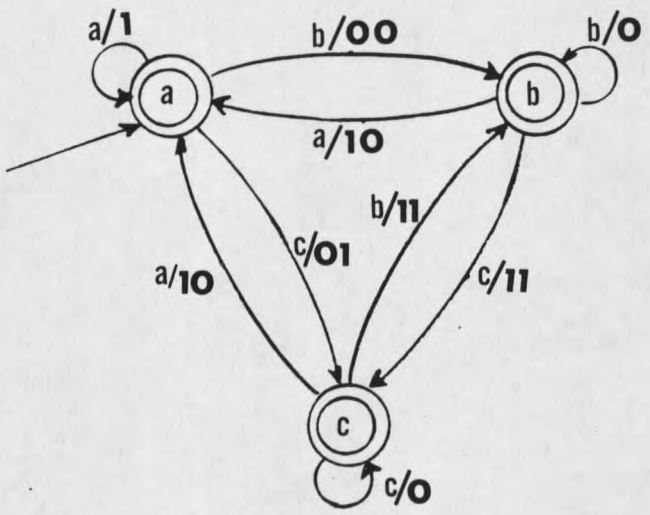


Figure 8. Encoding State diagram.

To encode, for instance, the source s = baabccccccc, the result will be 00101001100000. To decode, we use another state diagram as shown in figure 9.

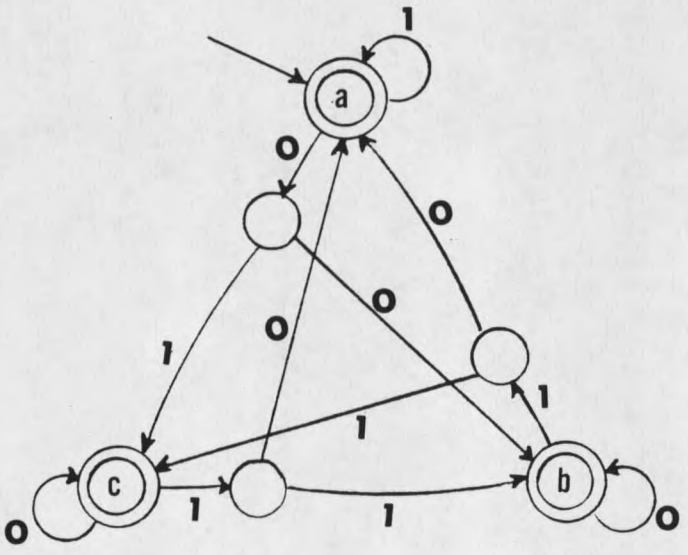


Figure 9. Decoding State diagram.

Now, let us calculate the efficiency of the Huffman code for the above Markov source. The entropy H(S) of a m^{th}-order Markov source [1] is

$$H(S) = \sum_{S^m} p(S_{j1},S_{j2},\ldots,S_{jm}) * \log (1/(p_{Si}|p_{j1},p_{j2},\ldots,p_{jm})) \quad (2.4)$$

The relevant probabilities are in table 3. The entropy is calculated using (2.4):

$$H(S) = \sum_{S} p(S_j, p_{Si}) * \log (1/(p(S_i|S_j))$$

$$= 1.42 \text{ bits /symbol.}$$

The efficiency is calculated using (2.1)

$$E = 1.42/1.55 = 0.92$$

Table 3. Probabilities of the Markov source in figure 7.

S_j	S_i	$p(S_i\|S_j)$	$p(S_j)$	$p(S_i,S_j)$
a	a	1/3	3/11	1/11
a	b	1/3	3/11	1/11
a	c	1/3	3/11	1/11
b	a	1/4	4/11	1/11
b	b	1/2	4/11	2/11
b	c	1/4	4/11	1/11
c	a	1/4	4/11	1/11
c	b	1/4	4/11	1/11
c	c	1/2	4/11	2/11

LZW Compression System

LZW compression algorithm [4] is a nonprobabilistic approach to data compression. The coding process is a string matching process which captures high usage patterns (strings) suggested in [9] and [10]. Contrary to the previous two Huffman codes, LZW is a typical fixed length code in which each code represents different numbers of source symbols.

Welch's modification [4] of Lempel and Ziv's idea in [9] and [10] makes it a lot easier to implement. We implemented the LZW data compression system according to the algorithm given in [4]. The results will be discussed in chapter 4.

The most important feature in LZW algorithm is the string table which holds all the strings we encounter in the source file. The only time that we output a code is when a particular symbol/string has been seen twice. The following is the main idea of the encoding process:

1. Initialize the string table to contain single character strings.
2. W ← the first character in the source file.
3. K ← the next character in the source file.
4. If the concatenation of WK is in the table,

   W ← WK,

   else

   output the code for W,

   put WK in the table,

   W ← K.
5. If not end of file, goto 3.

The basic idea of the decoding process can be simply as follows:

1. Initialize the string table to contain single character strings.
2. W ← decode the first code in the compressed file.
3. CODE ← the next code in the compressed file.
4. If CODE exists,

   K ← decode CODE,

   else

   K ← decode CODE as w_1W
5. W ← K.
6. If there are more codes, goto step 3.

Table 4 is the string table for the example in figure 10. The same table is re-produced by the decoding process in figure 11. The efficiency of the LZW code is a little harder to calculate because the code is a nonprobabilistic code. Though probabilities of symbols/strings are involved, they are not used in a direct sense. We know, however, that the lower bound of the code is the entropy by Shannon first theorem (1.4). Let us take the 12-bit code suggested by Welsh in [4] for instance. In the beginning of the encoding process, single symbols are represented by 12 bits while their information is

$$\begin{aligned} I(\text{single symbol}) &= -\log p(\text{single symbol}) \\ &= -\log 1/128 \\ &= 7 \text{ bits.} \end{aligned}$$

It is because the table is initialized with all the ASCII characters and there are 128 of them. The algorithm assumes them to be equally

Table 4. The LZW string table.

string	code
a	1
b	2
c	3
d	4
ab	5
ba	6
aa	7
aac	8
cd	9
da	10
aba	11
abab	12

W	K
a	b
b	a
a	a
a	a
aa	c
c	d
d	a
a	b
ab	a
a	b
ab	a
aba	b

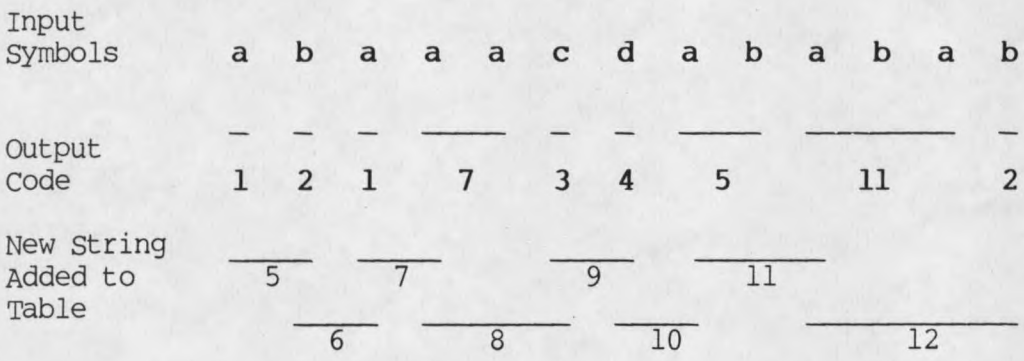


Figure 10. A compression example for LZW algorithm.

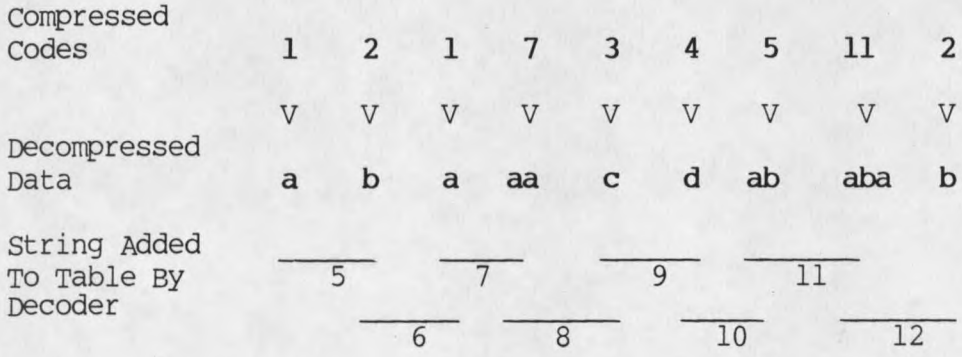


Figure 11. A decompression example of figure 10.

probable so their probability is 1/128. Since the LZW code and all the others we investigated are concatenation codes with integer code lengths, it leads us to believe that in order to construct a code with efficiency equal to one we have to use nonblock codes.

Universal Coder

Both the encoder and the decoder need the probabilities given by the modeler. Thus, a universal encoder needs only the probabilities to produce uniquely decodable codes. The universal decoder also needs the same probabilities to reproduce the source file. The Elias code [3] is the first code that tries to reflect the exact probabilities of the symbols/strings. Rissanen and Langdon in [5], [6], [10], and [11] generalize the concept of the Elias code to embrace other types of nonblock codes called arithmetic codes.

Idea Origin - Elias code

Elias code assigns each source sequence to a subdivision of the interval [0, 1]. To illustrate the coding process, consider a zero memory source with two symbols a and b, occurring with probabilities 0.7 and 0.3 respectively. We may represent an arbitrary infinite sequence of possible source file as figure 12. According to the figure, sequences starting with "a" are assigned to the interval [0, 0.7]; sequences starting with "aa" are assigned to the interval [0, 0.49]; and so on. To encode the sequence from the source we simply

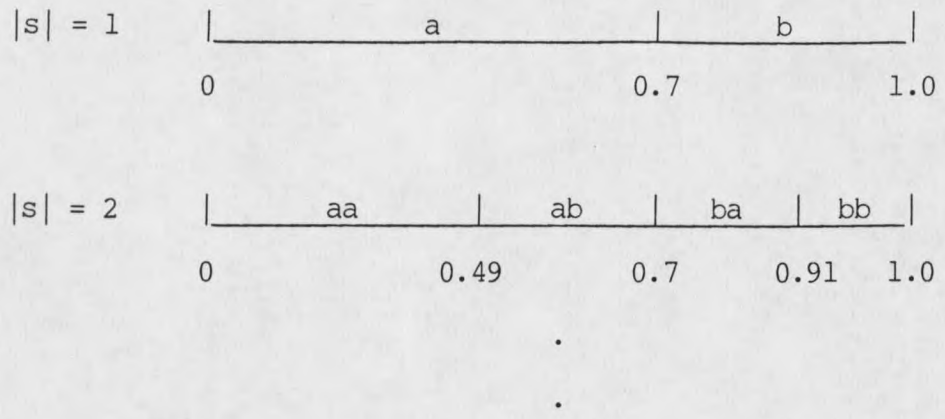


Figure 12. Subdivision of the interval [0, 1] for source sequences.

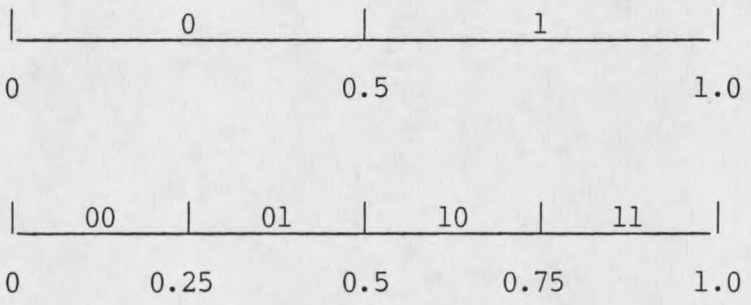


Figure 13. Subdivision of the interval [0, 1] with the probabilities in binary digits.

provide a binary expansion for each point in [0, 1] as in figure 13. Thus, a source that starts with "ba" will have acompressed file that starts with "10". Decoding is clearly a process of successive subtraction. If the binary sequence starts with "1011", the point represented must lie between 0.6875 and 0.75; hence the first three symbols from the source must be "bab".

The Algorithm

With the idea from Elias, we developed an arithmetic code which is similar to the one described in [10]. The difference is the way that we handle the precision problem and the carry-over problem.

The algorithm that we develop accepts any type of input source as long as we know the size of the source alphabet S, and the symbols are arranged in lexicagraphic or any fixed order. The basic idea of our code for any symbol S_j is the cumulative probabilities before S_j, $p(S_1) + p(S_2) + \ldots + p(S_{j-1})$. The code can be defined recursively by the equations

$$C(s\ S_1) = C(s),\ C(s\ S_i) = C(s) + P(S_{i-1}). \qquad (2.5)$$

Both the encoder and decoder require two binary registers, C and A. The former actually should be a buffer and the latter of width n (n will be discussed in the section precision). The codes are held temporarily in the C register, while the cumulative probability is held in the addent A. C represents the exact probability of the source file s with an imaginary decimal point in the front of the compressed file. The following is the encoding algorithm:

1. Initialization: C $\leftarrow$ all zero, Shift, Previous, LeftOver $\leftarrow$ 0.
2. $S_i \leftarrow$ read from the source file.
3. A $\leftarrow P(S_{i-1})$, Sht $\leftarrow \lfloor$ -log Previous$\rfloor$.
4. LeftOver $\leftarrow$ LeftOver - noninteger part of (-log Previous).
5. If LeftOver < 0 then LeftOver $\leftarrow$ 1 - LeftOver, Sht $\leftarrow$ Sht + 1.
6. Shift A by Sht bits from the end of C. ($|C| = 0$ initially)
7. C $\leftarrow$ C + A, Previous $\leftarrow p(S_i)$.
8. Goto 2 until the end of the source file.

The encoding process is basically shifting and adding. By the same token, the decoding process should be subtracting and shifting:

1. Initialization: Previous ← 1, LeftOver ← 0.

   C ← code from the compressed file.

2. i ← 2.

3. If C - Previous * $P(S_i)$ < 0

   then decode as symbol S_{i-1},

   Previous ← $p(S_{i-1})$,

   Sht ← |(-log Previous)|,

   LeftOver ← LeftOver - noninteger part of (-log Previous).

   If LeftOver < 0 then LeftOver ← 1 - LeftOver,

   Sht ← Sht + 1.

   Shift C Sht bits to the left.

   Goto (2).

   else i ← i + 1,

   Goto (3).

The C register, during the decoding process, has to have at least as many bits as the A register. To demonstrate the coding processes, consider S = { a, b, c, d } and s = abaaacdababab, the example that we used in figure 10. Assuming it is a zero-memory source, the statistics of s:

p(a) = 7/13,	P(a) = 0,
p(b) = 4/13,	P(b) = 7/13,
p(c) = 1/13,	P(c) = 11/13,
p(d) = 1/13,	P(d) = 12/13.

figure 14 is the encoding process of a simplier example when S = { a, b }, X = { 0, 1 }, s = a b b b b b b a a b ..., and p(a) = 1/8 and p(b) = 7/8. Figure 15 is the decoding process of the same example.

The Precision Problem

The algorithm in the last section omitted one very important thing — the number of bits that is required to represent A to ensure decodability.

Theorem: The number of bits that A needs is n, where n is the number of bits used in counting the occurrence of symbols/strings in the data compression system.

Proof: The smallest number between two addents is 1/n. To represent 1/n, we need exactly n bits in binary with an imaginary decimal in front.

Remarks: Even though n bits are used, the bits per code are by no means n because codes overlap each other unless the value of the addent is always 1/n. In order for that to happen, the previous symbol should always have the probability 1/n which is impossible unless $|S| = n = |s|$. Our idea is one of the ways of using floating point arithmetic to overcome the requirement for unlimited precision of the Elias code.

The Carry-over Problem

During the encoding process, the algorithms given in [6], [10], [11] , and the above may fail. When encoding a symbol with high

$S = \{ a, b \}$, $X = \{ 0, 1 \}$, s = a b b b b b b b a a b,
$p(a) = 1/8$, $p(b) = 7/8$, $P(a) = 000_2$, $P(b) = 001_2$.

Input	Register C	A	Shift	LeftOver
a	000	000	0	0
b	000001	001	3	0
b	0000011	001	1	1 - log (7/8)= .8074
b	0000100	001	0	.6147
b	0000101	001	0	.4421
b	0000110	001	0	.2294
b	0000111	001	0	.0368
b	00001111	001	1	.8441
a	00001111	000	0	.6515
a	00001111000	000	3	.6515
b	00001111000001	001	3	.6515

Compressed file: 00001111000001

Figure 14. The encoding process of the universal encoder.

Decoded as	Register C	Shift	LeftOver	Register C´
a	00001111000001	3	0	01111000001
b	01111000001	1	.8074	1011000001
b	1011000001	0	.6147	1001000001
b	1001000001	0	.4421	0111000001
b	0111000001	0	.1194	0101000001
b	0101000001	0	.0368	0011000001
b	0011000001	1	.8411	001000001
b	001000001	0	.6515	000000001
a	000000001	3	.6515	000001
a	000001	3	.6515	001
b	001	0	.4588	

Figure 15. The decoding process of the universal decoder.

probability, an addition is performed with a large number of 1's. If a long sequence of 1-bits has been shifted out of the C register, this will precipitate a "carry-over". If the sequence of 1-bits extends through the C register and into the already written code bits (in the compressed file), it becomes impossible to correctly perform the addition without rereading the compressed file.

Arbitrarily long sequences of 1-bits and therefore arbitrarily long carry-overs may be generated. In binary addition, the first 0-bit before a sequence of 1-bits will stop a carry-over. For this reason, inserting 0-bits into the code stream at strategic locations will prevent carry-overs from causing problems.

"Zero-stuffing" is used by Rissanen and Langdon [12] to check the output stream for sequences of 1-bits. When the length of one of these reaches a predefined value x, a 0 is inserted (stuffed) into the code stream just after the offending sequence. The sequence is undone by the decoder by checking x consecutive 1-bits and extracting the following bit of the sequence. In order that the decoder can correctly undo the zero-stuffing, 0-bits have to be inserted which are not necessarily needed for the carry-over problem. The way that we determine the value x can also cause problems. If x is too small, we could waste a lot of bits and consequently lead to poor compression. On the other hand, the size of x is restricted by the size of the C register. The worst thing about zero-stuffing is that we have to constantly check the C register for x consecutive 1-bits.

We use another solution by simply insert 0-bits at regular intervals in the code. Instead of using only the register for C, we

use an extra 4K buffer with the first bit (we actually use a byte) being the "carry-catcher". We always wait until the buffer is filled before we output it to the compressed file. Wasting one bit (byte) in 4096 saves a lot of time checking for x consecutive sequence of 1-bits. Moreover, long carry-overs are somewhat rare. Any that do occur tend to eliminate others from occurring because a long sequence of 1-bits turns to a long sequence of 0-bits when a carry ripples through.

This method works well. A carry-over only happens after a large addent with a small shift, it will not normally add any significant time to the coding process. Implementation is very simple. The insertions can be done with the normal file buffering - insert a 0-bit in the beginning of each buffer. The complexity of the process is further reduced by inserting an entire 0-byte in the beginning of each buffer. Byte-level addressing is often easier than bit-level addressing. The decoder then extracts and examines the entire byte rather than a bit.

Chapter 3

UNIVERSAL CODING WITH DIFFERENT MODELERS

Modeling an information source is without question the hardest part of any data compression system. The job of the modeler is to discover the redundancies or regular features in the data to be compressed. The information can be viewed as the probability distribution on the alphabet of the information source.

In order to use the universal coder, the coding process has to be divided into a more general form as in figure 4. The modeler is responsible for providing appropriate probabilities for both the encoder and the decoder. Even though some of the data compression systems stay away from the probabilities, like LZW [4] (an improvement of Lempel and Ziv's ideas in [9] and [10]) or the string matching techniques suggested by Rodeh [13], we can always convert their redundancy reduction techniques into our modeler. The conversion actually improves their codes compression-wise because we enforce our code to the lower bound of Shannon first theorem (1.4) with the appropriate probabilities.

Is There A Universal Modeler?

Since the universal encoder and decoder need only the probabilities of the symbols/strings and the probabilities of the previous symbol/string, the most basic modeler should provide these

probabilities. The easiest way to do that is to keep the counts of each symbol/string:

$$p(S_i) = \text{Count}(S_i) \;/\; \text{total count.} \qquad (3.1)$$

However, different information sources have totally different kind of redundancies. Welsh [4] suggested that there are four types of redundancies in commercial data, namely character distribution, character repetition, high-usage patterns and positional redundancy. To capture a particular kind of redundancy, we need to analyze the information source to decide what kind of redundancy capturing techniques have to be used. For instance, the probabilities we need in a first order Markov source are $P(S_x|S_y)$, which can be given by

$$p(S_x|S_y) = \text{Count}(S_x|S_y) \;/\; \text{Count}(y) \qquad (3.2)$$

where Count(y) denotes the number of times class y occurs in the source s, and $\text{Count}(S_x|S_y)$ denotes the number of times the next symbol at these occurrences is S_x.

Modelers can be divided into two categories, namely nonadaptive and adaptive. The following sections show how we can convert existing models to provide appropriate probabilities for the universal coder so as to improve compression.

Nonadaptive Models

Nonadaptive models basically scan the information source to get the appropriate probabilities. The probabilities will be put in the model file so that both the encoder and decoder can use. Therefore, the decoding process cannot be performed with only the compressed

file, and the compression ratios defined in (2.2) should be redefined as

$$PC = (|OF| - |CF + MF|) / |OF|. \qquad (3.3)$$

The major disadvantage of nonadaptive models is the fact that they require two passes of the information source before the compressed file can be generated. Nevertheless, the probabilities provided by a nonadaptive model is usually better than an adaptive model, therefore better compression can be expected if we can keep the size of the model file down.

Zero-memory Source

For a zero-memory source, we have been using the Huffman code since the fifties. However, it has been shown by Shannon's first theorem that the worst a code would do is one extra bit more than the entropy. Though the Huffman code does not separate the modeling process from the coding, we can view the first step of the following algorithm of the Huffman code as modeling and the rest as the encoding process:

1. scan the source file by counting the occurrence of each source symbol.
2. arrange the symbols in descending order with respect of their probabilities.
3. perform the reduction and splitting process as in figures 5 and 6 to generate the code.
4. build a finite state machine as in figure 8.
5. encode the source file using the machine built in step 4.

For the universal coder, we need the probabilities of each symbol just as in the above algorithm. The modeler is responsible for providing (2.1) for all i by simply counting the occurrence of each symbol. In other words, the modeler for any zero-memory source is step 1 of the algorithm above. For example, S = { a, b, c, d, e }, s = a b a c d b a e c a, and the model file can be represented by

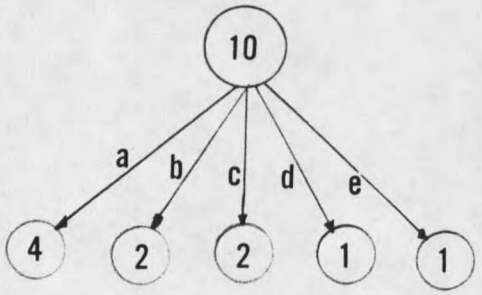


Figure 16. An example of a model file.

**The model file above actually represents the statistics for figure 6.

Markov Source

To design a data compression system for a Markov source, the modeler has to provide the probabilities given in (3.2). From (3.2), we can calculate the information content of the source by

$$I(S) = \sum_{z=1}^{K} \text{Count}(z) \log \text{Count}(z) - \sum_{z=1}^{K} \sum_{x \in S} \text{Count}(x|z) \log \text{Count}(x|z). \quad (3.4)$$

Proof: Since $p(x|z) = \text{Count}(x|z) / \text{Count}(z)$, and

$$p(s) = \prod_{x \in S} \prod_{\text{Count}(x|Q_1)} p(x|Q_1) \cdots \prod_{x \in S} \prod_{\text{Count}(x|Q_K)} p(x|Q_K). \quad (3.5)$$

$$1/p(s) = \prod_{x \in S} \prod_{\text{Count}(x|Q1)} \text{Count}(Q_1)/\text{Count}(x|Q_1) \cdots \prod_{x \in S} \prod_{\text{Count}(x|Qk)} \text{Count}(Q_K)/\text{Count}(x|Q_K)$$

$$I(s) = \sum_{x \in S} \sum_{Count(x|z)} \sum_{z=1}^{K} \log Count(z) - \sum_{x \in S} \sum_{Count(x|z)} \sum_{z=1}^{K} \log Count(x|z)$$

$$= \sum_{z=1}^{K} \log Count(z) \left[\sum_{x \in S} \sum_{Count(x|z)} 1 \right] - \sum_{z=1}^{K} \sum_{x \in S} Count(x|z) \log Count(x|z)$$

$$= \sum_{z=1}^{K} Count(z) \log Count(z) - \sum_{z=1}^{K} \sum_{x \in S} Count(x|z) \log Count(x|z).$$

Let us illustrate nonadaptive models with the example we used before where the source s = abbbaccbaaabbbcccabcaac, and the source alphabet S = { a, b, c }. By assuming the source to be a first-order Markov source, the modeler counts the symbols conditioned on the previous symbols. With the initial state as **a** we should get the conditional count

Count(a|a) = 3 Count(b|a) = 3 Count(c|a) = 3

Count(a|b) = 2 Count(b|b) = 4 Count(c|b) = 2

Count(a|c) = 2 Count(b|c) = 2 Count(c|c) = 4

to produce the model file in figure 17. By (3.4), the ideal code length with this model is

I(s) = 9log9 + 2 * 8log8 - (3 * 3log3 + 4 * 2log2 + 2 * 4log4)

= 38.26 bits.

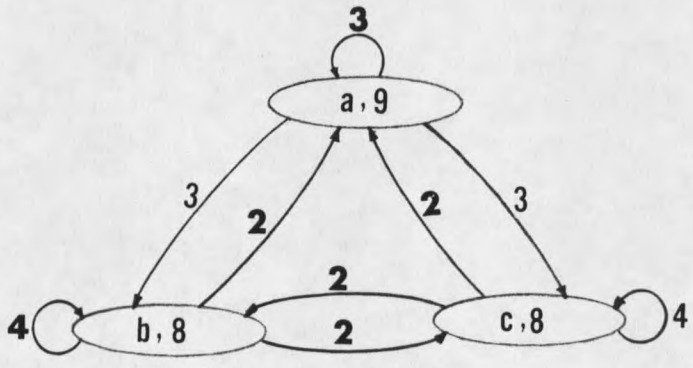


Figure 17. An example of a model file for a first-order Markov source.

Adaptive Models

The major disadvantage of the nonadaptive models is the fact that the information source has to be prescanned by the modeler to get the appropriate probabilities. More importantly, the model file containing the probabilities has to be transmitted to the decoding units as a preamble in the code string. Since the model file is not needed in the data compression system using an adaptive modeler as shown in figure 4, compression could sometimes be better especially for Markov source where the model file is large. In a first-order Markov source, the model file contains $|S|^2$ probabilities. When the number of orders goes up, the number of probabilities in the model file goes up exponentially,

$$\text{Number of probabilities needed} = |S|^{\text{order}}. \qquad (3.6)$$

An ASCII file which is a third-order Markov source requires $256^3 = 16777216$ probability values. To represent a higher order of Markov source, the model file could even be bigger than the source file.

To avoid prescanning the source and the preamble, we need to consider adaptive models in our data compression system. As a matter of fact, adaptive models also give the modeler a chance to adjust the probabilities as the source is being encoded.

Zero-memory Source

The Huffman code we looked at earlier is a classic example of a code using a nonadaptive model. The conversion of such a nonadaptive model to an adaptive one is quite straight forward. Equation (3.1)

still holds while the probabilities are calculated after each symbol is scanned. The modeler of an adaptive model does the followings:

1. initialize the source symbols to be equally probable.
2. read in a symbol from the source file, and send the present probabilities to the encoder.
3. update the probabilities with the current source symbol.
4. If not end of file, goto step 2.

Equation (3.1) has to be recalculate after each symbol is scanned because both the total count and the count of the present symbol change. Figure 18 is what an adaptive modeler should provide to the encoder and decoder.

Markov Source

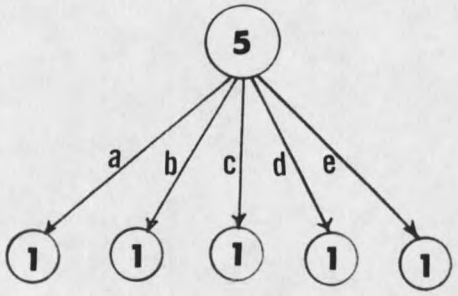
The adaptive modeler for a Markov source is not much different than a zero-memory source, as each state of the finite state machine of the Markov source is the same as the zero-memory source. Combining figure 17 and figure 18, figure 19 is an example of a first-order Markov source.

LZW Code

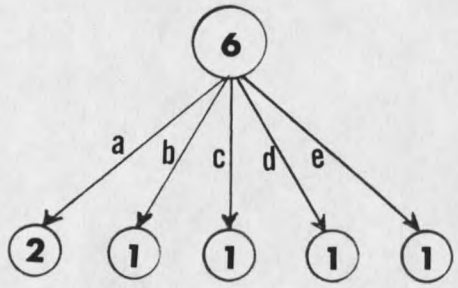
Though the LZW compression algorithm [4] does not seem to fit our universal coder because it is a nonprobabilistic approach, we can use their idea in building such a modeler that provides probabilities of symbol/string for the encoder and the decoder. We can consider the string table generated by the LZW algorithm as the modeler which

S = { a, b, c, d, e }, s = a b a c d b a e c a.

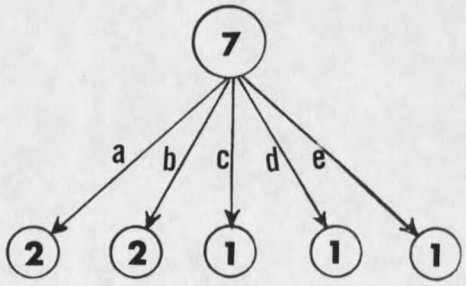
Before scanning,



After scanning **a**,



After scanning **b**,



After scanning the source s,

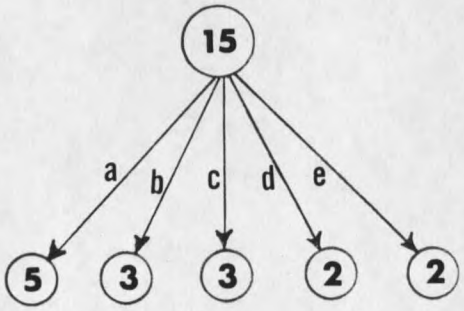
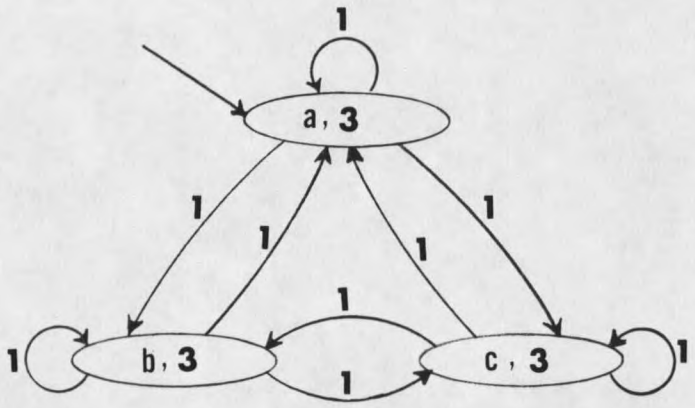


Figure 18. An example of an adaptive modeler for zero-memory source.

S = { a, b, c }, s = abbbacccbaaabbbcccabcaac.

Before scanning,



After the source s,

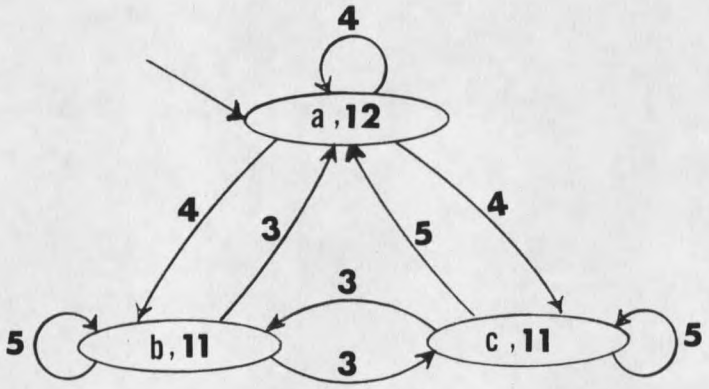


Figure 19. An example of an adaptive modeler for a first-order Markov source.

counts the symbols and strings. We can build the modeler with a tree keeping the count of all symbols and strings that the modeler has scanned. We encode one symbol at at time by parsing the tree we built. We build the LZW modeler as follows:

1. Initialize the count of lambda to q ($|S|$), and the count of all the symbols to one. Initialize the pointer W to Lambda. Previous ← 1.
2. Read one symbol S_i form the source file, send $P(S_i)$ and Previous to the encoder.
3. Previous ← $p(S_i)$ / W, Update W.
4. Move W to point to S_i.
5. Read one symbol S_j from the source file,

   if S_j is a son of W

   then send $P(S_j)$ and Previous to the encoder.

   Previous ← $p(S_j)$ / W.

   Update W.

   move W to S_j.

   else

   send $P(S_j)$ and Previous to the encoder.

   Previous ← $p(S_j)$ / W.

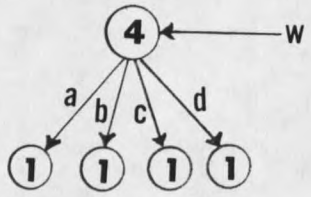
   Update both W and p_j.

   Move W to lambda.

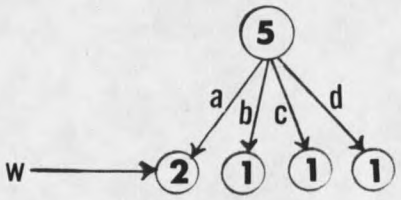
Figure 20 is the example in table 4 by using such a modeler. The beauty of such a modeler is it converts a nonprobabilistic algorithm

S = { a, b, c, d }, s = a b a a a c d a b a b a b.

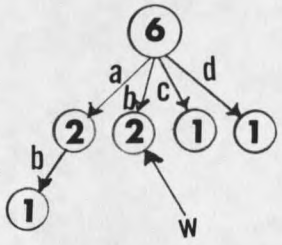
Before scanning,



After scanning **a**,



After scanning **b**,



After scanning the source s,

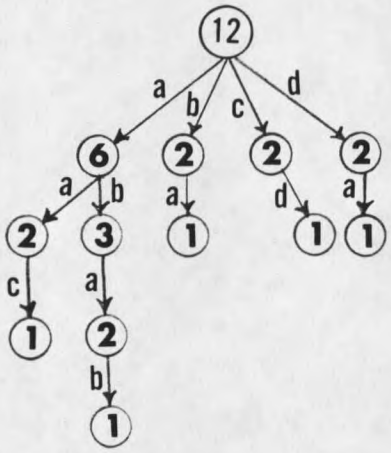


Figure 20. An example of converting the LZW table into a modeler.

into a probabilistic one while preserving the same process of capturing redundancies. The major shortcoming of using such a modeler is the speed of both the encoding and decoding processes. Each time a symbol is encoded, we have to go through the whole process of calculating all the probabilities at one level. Since the LZW algorithm uses an adaptive modeler, the decoding unit simply uses the same modeler and the decoder described in chapter 2.

Chapter 4

COMPRESSION RESULTS AND ANALYSIS

Test Data and Testing Method

The test data for the universal coder with different modeler consisted of different types of redundancies -- English text, gray-scale images, and floating point arrays. The floating point arrays we used are files which do not contain any "natural" redundancies. We use 0, 1 and 2 Taylor Series approximation (predictor), otherwise no compression is expected no matter what code we use. Whenever a predictor is used, we use a particular predictor throughout one test of a particular file to compare the results.

As we have discussed in the previous chapter, building a model for a high order Markov source is too expensive memory-wise. We test the compression of our universal coder and different modelers by compressing the same file with Huffman code and LZW code, and then with the universal coder with zero-memory modeler and with LZW modeler.

Table 5 shows the compression results of different codes with and without using the universal coding scheme. We can see the clear improvement in compression; the speed trade-off will be discussed in section model analysis.

Table 5. Compression results for a variety of data types.

			with our modeler and universal coder	
Data Type	Huffman	LZW	Zero-memory	LZW
English text	25%	45%	33%	55%
Floating Arrays	5%	10%	28%	30%
Gray-scale images	5%	35%	30%	50%

Code Analysis

The coding unit is based on the idea of the Elias code [3]. The code produced by the universal coder that we built can be called as Arithmetic Code which in its original practical formulation is due to J. Rissanen [12]. We also constructed the Fast Arithmetic Code [10] which is an implementation of Arithmetic Code for sources using binary alphabets and which uses no multiplication. The price paid for the resulting faster coding is a loss of at most 3% in compression performance. In practice, the Fast Arithmetic Code may be a better code to use because it is about twice as fast as our universal code. However, the Fast Arithmetic Code can only handle binary source code so for most cases the universal code has the clear advantage.

There are also several advantages to using the universal code as opposed to other available codes. First, the universal code can be

used with any model of the source. Other codes have been historically highly model dependent and in fact the coding and the model are hard to separate. The universal code enables the designer of a data compression system to emphasize the modeling of the data. Modeling is where the difficulty in data compression really lies because of the variety of redundancies.

The adaptability of the universal code allows the modular design of a data compression system. If any changes are made to the system, for example changing from floating point numbers to gray scale images or from a nonadaptive approach to an adaptive one, only the modeler needs to be changed and not the coder. This flexibility allows for the design of one data compression system which is able to compress various types of data under various circumstances without the need of rebuilding the system for each type data. It also allows for models to use any type of source alphabets.

The most significant gain in the universal code is that it is a noninteger code and so can perform compression as low as the entropy. It is then easier for us to compare one model to another.

Model Analysis

Since our universal coder guarantees to generate codes with the shortest average code length, we can concentrate on finding appropriate models for particular sources. Remember the definition of the entropy -- the best possible compression with the **given** probabilities. The word "given" is the key of a modeler. The several models we discussed in the previous chapter provides totally different

kinds of probabilities. It is not possible to conclude which is the best, but rather which is better for a particular source. Intuitively, it is safe to say that LZW model works best for most sources because LZW works like a variable order Markov process. Only the popular patterns are put in the string table. Before we study the best and worst case in data compression, let us consider a general information source.

Let the source alphabet $S = \{S_1, S_2, \ldots, S_q\}$, the information source $s \in S^*$, and s is made up of $w_1, w_2, \ldots, w_i, \ldots, w_m$, where $w_i \in S$ and $|s| = m$.

The Best Case

The best case for any data compression system is when the file contains only the repetition of a single symbol of the source alphabet.

First of all, let us look at how nonadaptive model would do. In a zero-memory source or Huffman code, a single symbol S_x will have the probability equal one. S_x will be represented by one bit in Huffman code. The percentage compression of Huffman code by (2.2) is $PC_{Huffman} = 1 - \lceil$(m + # of bits in the model file) / m * # of bits per source symbol$\rceil$, so

$PC_{Huffman} \sim 1 - (1 /$ bits per source symbol).

By using the universal coder and the zero-memory modeler, the encoding unit should simply send the model file and the number of occurrences of symbol S_x to the decoder. This is exactly the way that a run-length code would handle the situation. Thus the compression is

even better than the Huffman code though the theoretical 0 bit is needed to encode such a source, because

$$H(s) = 1 \log 1 + (q - 1) * 0 \log 0$$
$$= 0.$$

If we treat such a source as a Markov source, we should come up with the exact result except a bigger model file. It is because both the encoding and decoding process stay in one state of the finite state machine as in figures 8 and 9.

For adaptive models, the performance of both zero-memory sources and Markov sources are somewhat worse than the nonadaptive models in the best case. It is because we start off with equal probabilities of all symbols. It is interesting to see how the LZW modeler performs under the situation. By the LZW algorithm, the source $s = w_1, w_2, .., w_i, .., w_m$, ans s is partitioned into $t_1, .., t_i, .., t_r$, where t_i S^*. The encoded file of s is the concatenation of the code of $t_1, ..,$ and t_r, that is to say $C(s) = C(t_1)\ C(t_2) \ldots C(t_i) .. C(t_r)$. For the best case, $w_i = w_j$, for i, j = 1, 2, ..., m. The partition is $t_1 = w_1$, $t_2 = w_2w_3$, $t_3 = w_4w_5w_6$, ..., and $|t_1| = 1$, $|t_2| = 2$, $|t_3| = 3$, and so on. For instance, S = { ASCII characters}, and s = aaaaaaaaaaaaaaa. |S| = 128 and |s| = 15. s is partitioned into a, aa, aaa, aaaa, aaaaa, and the code of s C(s) = C(a) C(aa) C(aaa) C(aaaa) C(aaaaa). Thus s can be encoded by 5 codes. In general, if s = |m|, and the number of code needed to encode s is N,

$$\sum_{i=1}^{r} i \ >= \ N \ > \ \sum_{i=1}^{r} i$$
$$r(r - 1) / 2 \ >= \ N \ > \ (r - 1)(r - 2) / 2$$
$$N - \lceil \text{sqrt}(2m) \rceil \qquad (4.1)$$

If we use a 12-bit code as suggested in [4], the compressed file is about sqrt(2*m) * 12 bits. By using the LZW modeler and the universal coder, we need (7 + 12) / 2 bits for each code on the average. In other words, we can save 2.5 bits per code and the number of code needed is sqrt(2*m).

The Worst Case

The worst case for any data compression system is a file with all the symbols and possible strings having exactly the same probabilities.

For nonadaptive models, $p(S_i) = p(S_j)$ for $i,j = 1, 2, \ldots, q$ in zero-memory source, and $p(S_j|Y_i) = p(S_j)$ for all j and $Y_i = 1, 2, \ldots, q$. The best that the universal coder using a zero-memory modeler or a Markov source modeler is

$$H(s) = q * 1/q \log q. \qquad (4.2)$$

$$I(s) = m * H(s). \qquad (4.3)$$

For adaptive models, equation (4.2) and (4.3) still hold because an adaptive model starts with equal probable symbols and the count of symbols and strings never changes much in the worst case.

The original LZW algorithm, however, produces a compressed file which is bigger than the source file because the nature of the fixed length code. It is the reason why the LZW algorithm gives negative compression (expansion) in some cases.

Conclusions

Different models produce different ratios of compression on different source files. However, the universal coding scheme

described in this paper guarantees the shortest average code length for the given probabilities. More importantly, the universal coder can be used with any model of the source, and the conversion of most data compression systems into modeler and coder is quite easy and straight forward.

Nevertheless, using an adaptive model like LZW with the universal coder slows down the speed of the whole coding process by as much as 50 times comparing with the original LZW code while 10 to 20 percent extra compression is gained on the average. It is up to the user to decide if compression is more important than the speed of the coding process. It also leads to future research in speeding up the coding process for adaptive models which are supposed to be faster than nonadaptive models.

Future Research

Future research in the whole area of data compression may be concentrated on modeling. Since no universal modeling technique can guarantee effective compression, it would be nice if there existed an analyzer which determines which modeler works best for any given source file.

Data compression systems which deal with digitized images complicate the problem even further because some images do not possess any "natural" redundancies. The modelers we look at in this paper capture only sequential redundancies while in most images, the value of a pixel depends on the surrounding pixels. Predictors described in [14], for instance, have to be used.

Furthermore, adaptive models using the universal coder performs quite a bit slower than nonadaptive models because of the recalculation of probabilities after each symbol has been scanned. However, parallel algorithms seem to fit very well with the probability calculations. Speed is expected to increase dramatically if such algorithms are employed.

REFERENCES CITED

REFERENCES CITED

[1] Hamming, Richard W. Coding and Information Theory, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[2] Rivest R.L.; Shamir A.; and Adleman L. "On Digital Signatures and Public Key Cryptosystems", Communications of the ACM, Vol. 21, No. 2, February 78, p. 267-284.

[3] Abramson, Norman. Information Theory and Coding, McGraw-Hill Book Company, New York, New York, 1963.

[4] Welch, Terry A. "A Technique for High-Performance Data Compression", IEEE Computer, June 1984, p. 8-19.

[5] Langdon, Glen G. "Antroduction to Arithmetic Coding", I.B.M. J. Res. Develop., Vol. 28, No. 2, March 1984, p. 135-149.

[6] Rissanen, Jorma; and Langdon, Glen G. "Arithemetic Coding", IEEE Transaction on Information Theory, Vol. 27, No. 1, March 1979, p.149-162.

[7] Campell, Graham. "Two Bit/Pixel Full Color Encoding", ACM SIGGRAPH 1986 Proceedings, Vol. 20, No. 4, 1986, p. 127-135.

[8] Regbati, K. "An Overview of Data Compression Techniques", IEEE Computer, April 1981, p. 71-75.

[9] Lempel, Abraham; and Ziv, Jacob. "On the Complexity of Finite Sequences", IEEE Transaction on Information Theory, Vol. 22, No. 1, January 1976, p. 75-81.

[10] Langdon, Glen G. "A simple General Binary Source Code", IEEE Transaction on Information Theory, Vol. 28, No. 5, September 1982, p. 800-803.

[11] Langdon, Glen G.; and Rissanen, Jorma. "Compression of Black-White Images with Arithmetic Coding", IEEE Transaction on Information Theory, Vol. 29, No. 6, June 1981, p. 858-867.

REFERENCES CITED--Continued

[12] Rissanen, Jorma; and Langdon, Glen G. "Universal Modeling and Coding", IEEE Transaction on Information Theory, Vol. 27, No. 1, January 1981, p. 12-23.

[13] Rodeh, Michael; Pratt, Vaughan R.; and Even, Shimon. "Linear Algorithm for Data Compression via String Matching". Journal of the Association for Computing Machinery, Vol. 28, No. 1, January 1981, p. 16-24.

[14] Henry, Joel E. "Model Reduction and Predictor Selection in Image Compression", Master's thesis, Department of Computer Science, Monatan State University, Bozeman, Montana, August 1986.

APPENDICES

APPENDIX A

Glossary of Symbols and Entropy Expression

General Symbols

S	source alphabet
S_i	source symbol
$q = \|S\|$	number of source symbols
S^n	nth extension of source alphabet
s	information source, an element of S^*
$\|s\|$	size of an information source
$C(S_i)$	code of the source symbol S
$C(s)$	code of the inormation source s
X	code alphabet
X_i	code symbol
r	number of code symbol
l_i	number of code symbols in code word X_i corresponding to S_i
L	average length of a code word for S
L_n	average of a code word for S_n

Entropy Expressions

$$L = \sum_{i=1}^{q} |C(S_i)| \, P(S_i)$$ the average code length

$I(S_i) = - \log P(S_i)$	he amount of information obtained when S_i is received
$H(S) = -\sum_S P(S_i) \log P(S_i)$	entropy of the zero-memory source S
$I(s) = \|s\| * H(S)$	amount of information of an information source

APPENDIX B

Definitions of Codes

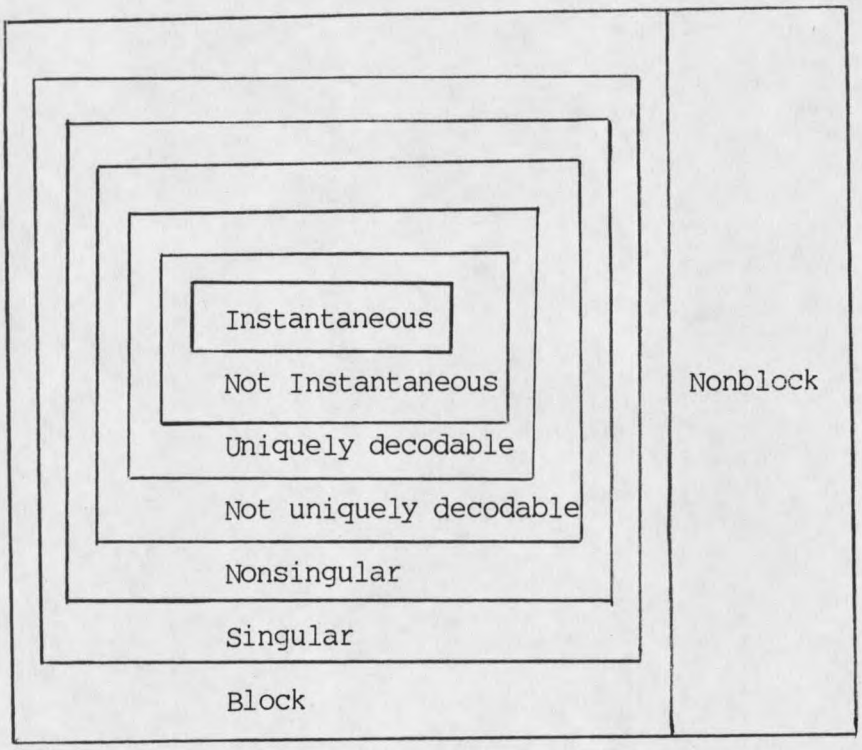


Figure 21. Subclasses of codes.

Definitions:

A **block code** is a code which maps each of the symbols of the source alphabet S into a fixed sequence of symbols of the code alphabet X.

A block code is said to be **nonsingular** if all the words of the codes are distinct.

The n^{th} extension of a block code which maps the symbols S_i into the code words X_i is the block code which maps the sequences of source symbols $(S_{i1}S_{i2} \cdots S_{in})$ into the sequences of code words $(X_{i1}X_{i2} \cdots X_{in})$.

A block code is said to be **uniquely decodable** if, and only if, the n^{th} extension of the code is nonsingular for every finite n.

A uniquely decodable code is said to be **instantaneous** if it is possible to decode each word in sequence without reference to succeeding code symbols.