NEAR EAST UNIVERSITY

1988

COMPUTER ENGINEERING DEPARTMENT

GRODUATION PROJECT COM 400 ENTROPY CODING

SUPERVISOR :
FAHRETTIN M. SADIGOGLU

MUSTAFA DINC
92304

MUTLU SAYAR
93078

JUNE 1997

TABLE OF CONTENTS

INTRODUCTION

First of all we should deal with what a source coding is. In *figure1* it is shown a schematic diagram of a transmission system. It contents most of the components of a transmission system.

A trnsmission system transmits signals from a source to a destination, which sometimes called a sink. The source is the device by which the signals are generated. The destination is the place where they are received.

The central part of the transmissin system is the channel. This is the medium which physically transports the signal, for example electrical sygnal, as an electromagnetic wave or as a sequence of light pulses. Practical examples of channels are a coaxial cable, an optical fiber and the atmosphere. Apart from the channel, the sygnal passes through three pairs of blocks: the modulator and the demodulator blocks, the error protection and correction blocks, and the source encoder and decoder blocks. These pairs of blocks are present because of three channel properties discused below.

The first channel property is that channels can only transport physical (e.g. electrical) signals. For successful transmision a digital sygnal must therefore be converted to a form appropriate for the channel. For radio transmission, for instance, the signal must be modulated on a carrier and then converted into an electromagnetic wave. The modulator converts the signal to a representation that is appropriate for the channel. The demodulator converts the received signal back to its original form.

The second channel property is that, even if a signal is adapted to the channel, it does not always pass it undisturbed: the channel may introduce errors. If digital information is transmitted, the result can be that after demodulation some bits are inverted. The block before the modulator is the error-protection unit. It adds bits to the signal that

make it possible for the error-correction unit to detect and even correct errors introduced by the channel.

The third channel property is that there is an upper bound to the number of bits per second that can be correctly transmitted. This bound is called the channel capacity. The source-encoding block reduces the number of bits per second with which the input signal is represented, to a number that is low enough for transmission. The signal with the reduced bit rate is the source-encoding signal. The source decoder converts this to a reconstruction of the input signal. Unfortunately, source encoding and decoding may change the signal. This results in the reception of a distorted signal. In a good source-coding system the distortion is kept below a certain level.

It is mainly source coding for speech, music and pictures that is considered here. This implies that one finds a human observer at the destination. This has its impact on the notion of distortion and on the design of source-coding systems. Agood source-coding system keeps distortion below a certain level. If signals such as speech, music and pictures are received by a human observer, it means that after reception these signals must have a desired subjective quality rather than a desired objective quality.

In most of what follows it is assumed that the concentrantion of the error-protection block, the modulator, the channel, the demodulator and the error-correction block behaves as a digital, error-free channel, which imlies that the source decoder receives the undestorted output oh the source encoder.

Source coding is not only the name for the discipline involved with the design of source-coding algorithms and systems but also for the action of the source encoder and decoder. Other names that are sometimes used for the source coding are bit-rate reduction, data reduction and data compression. The combination of a source encoder and decoder is often called codec.

Examles of source coding applied in transmission and storage systems are: source codig of speech signals in mobile automatictelephony, source coding of x-ray and nuclear magnetic resonance images for storage in medical databases, source coding of sound signals for storage on compact disc interactive (CD-I) disks and on digital compact cassette (DCC) tapes and for digital audio broadcasting, source coding of images of documents for storage in the Megadoc system, and source coding of digital TV pictures for storage on a digital video tape.

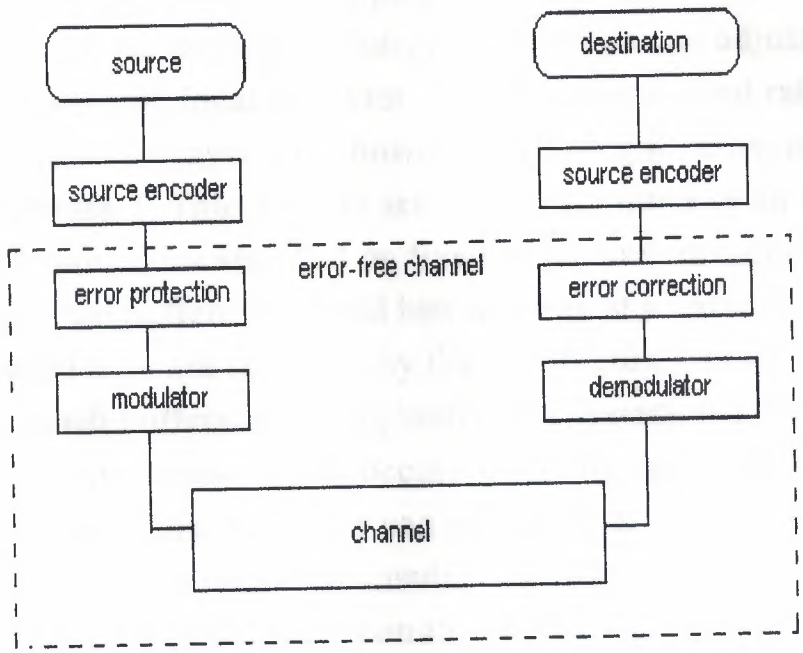

Figure1

ENTROPY CODING

Most of the coding systems are *fixed rate codes* in the sense that a fixed number of channel bits per time unit is produced by the encoder and processed by decoder. Examples of these type of codes are quantization, bit allocation and transform coding. In some communication and storage systems, fixed rate operation is not desirable because the data source may display wide variations of activity. For example, samled speech may change very little during long periods of silence and then exhibit very complex behavior during plosives. Ideally, one would like to waste few bits coding the silence and preserve them for coding the highly informative transitients. Such a strategy requires a *variable rate code*, a code which can adjust its own bit rate to better match local behavior. In order to use fixed rate communication and storage links, however, the long term average bit rate must be constant. Thus buffers are usually required as an interface when variable rate codes are used on fixed rate communication or storage media. The buffers will hold bits arriving at a variable rate from the encoder until they are accepted by the fixed rate channel for transmission. Such buffers add complexity to a system and can also add errors when they overflow, which occurs when the data source produces bits faster than the buffer can accept them. Similarly, errors can be introduced when the buffers underflow, which occurs when the data source produces bits slower than the rate at which the buffer is releasing bits. To combat this problem, a technique known as *buffer feedback* is commonly used, where the occupancy level of the buffer is fed back to the source encoder to suitably adjust the quantizer data rate. This added complexity is often justified, however, by the potentially significant performance gains possible with the variable rate strategies. Entropy codes are often used in conjunction with scalar quantizers (to conserve the average bit rate) and are often fairly simple to implement when the input alphabets are of reasonable size.

The overall variable rate code is then a simple cascade of a scalar quantizer, which performs the analog-to-digital conversion in a fixed rate manner, and a variable length noiseless code, which maps the quantizer output into a variable length binary index in a way that can be perfectly decoded by the receiver.

Communication and storage systems that are inherently variable rate are increasing in importance and variable length codes can be well matched to such systems. For example, variable rate codes cause no problems in offline starage (the bits are accepted as they come until the file is complete) and variable rate codes are no more complicated than fixed rate codes for use in packet communication environments.

Entropy coding is also often referred to as *noiseless coding, lossless coding,* and *data compaction coding.* It is also referred to simply as *data compression* in the computer science literature, but it is avoided that this nomenclature as entropy coding is a very special case of data compression. The narrow use of the term by computer scientists is perhaps understandable because of the disastrous consequences that can result from even rare bit errors if the compressed file is a binary executable file. When bit errors cause catastrophe, lossy codes are not useful for compression (except possibly as a component of an overall lossless code).

The goal of noiseless coding is to reduce the average number of symbols sent while suffering no loss of fidelity. A classical example is the Morse code where short binary codewords are used for more probable letters and long codewords used for less probable letters. The Morse code in fact is a very good code for its age and, when applied to English text, results in many fewer bits on the average than would the use of one byte ASCII codes for each letter. A more recent but still venerable example is the run-length code used to code sources which tend to repeat symbols for long periods of time. For example, a binary source such as facsimile may produce long runs of zeros and occasionally, ones. Hence one means of compression is to sequentially

send a symbol followed by the number of its repetitions, the *run lenth.* This will result in compression on the average if the source tends to produce such runs. It will not compress a memoryless source.

Variable-Length Scalar Noiseless Coding

note: In my report I tried to avoid using mathematical expressions but I used the ones that are unavoidable for explaining the event.

Suppose that $\{ X_n \}$ is a stationary sequence of random variables with a finite alphabet $A = \{ a_0, \ldots, a_{M-1} \}$ with a marginal probability mass function $p(a) = p_x (a) = \Pr (X_n = a)$. The case of of primary interest for the present purposes is that where the X_n are quantized versions of continuous alphabet sequence W_n , that is, $X_n = Q (W_n)$, with q an ordinary scalar quantizer.

A *variable length scalar noiseless code* consists of an encoder α , which maps a single input symbol χ in A into a binary vector $\alpha (\chi)$ of dimension or length $l(\chi)$, and a decoder β , which maps binary vectors u of differing length into an output $\beta (u)$ so that $\beta (\alpha (\chi)) = \chi$; that is, the encoding / decoding operation is *lossless* or *noiseless* or *transparent*. The goal of the code is to keep the average number of bits transmitted for each source symbol as small as possible, that is, to minimize the *average length*

$$l (\alpha) = E\, l (X_n) = \sum_{A \varepsilon A} p (a)\, l (a).$$

formula 1.

If *formula 1* is accepted as a definition of quality of noiseless source code, then it is of interest to quantify how small $l (\alpha)$ can be made and hence what the optimal achievable performance is. It is also

of interest to construct actual codes that perform very near to the optimal quantity.

Unfortunately, the given definition of a code is not enough to ensure that it is useful. Suppose, for example, that the input alphabet has 4 letters,

$A = \{ a_0, a_1, a_2, a_3 \}$, possibly the output of a 2 bit per sample quantizer.

Input letter	Codeword
a_0	0
a_1	10
a_2	101
a_3	0101

table 1.

Although this is a noiseless code by the above definition, it cannot always be decoded in a noiseless fashion when the code is applied to a sequence of inputs. For example, if the receiver gets the sequence 0101101.... , it could have been produced by the input sequence $a_0a_2a_2a_0a_1$.... or by $a_3a_2a_0a_1$... . To make matters worse, the ambiguity can never be resolved regardless of future received bits. Hence for a code to be useful, it must be *uniquely decodable* in the sence that if the decoder receives a valid encoded sequence of finite length, there is only one possible input sequence that could have produce the encoded sequence. The effectively extends the idea of a noiseless or transparents code from a single letter to a sequence. Note that we could accomlish this by inserting punctuation in the binary sequence between codewords, e.g., add a third letter "," and send the sequence 0, 101, 101, 0, 10,... While this disambiguates the sequence, it also increqses the average length of the encoding as well as the required channel alphabet. This may be a simple fix, but it is not an efficient use of symbols. An alternative and less restrictive approach is to require that the code satisfy a *prefix condition* in the sense that no codeword be a prefix of any other codeword. In the previous example, a_0 is a prefix of

a_3 and a_1 a prefix of a_2. An example of a code satisfying the prefix condition is given below.

Input letter	Codeword
a_0	0
a_1	10
a_2	110
a_3	111

table 2.

Binary prefix codes can be depicted as a binary *tree* as below.

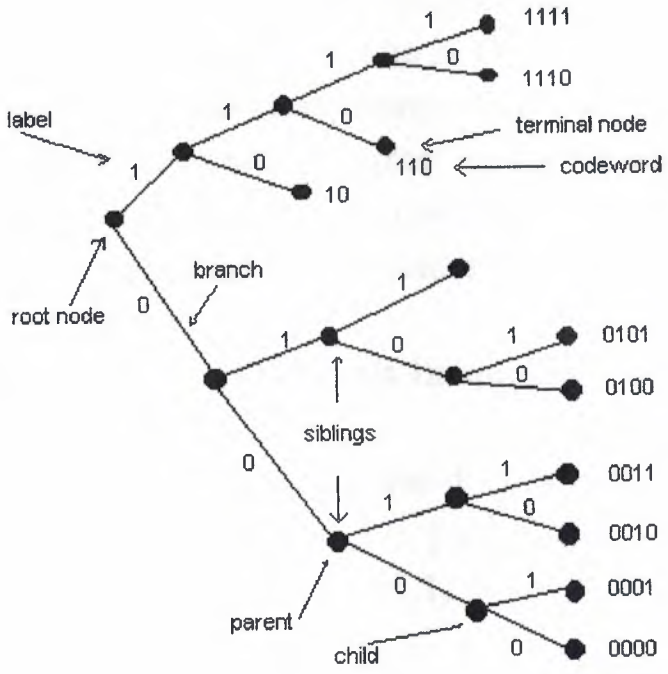

figure 2.

The binary tree starts with a *root* or *root node* which has *branches* extending from it. Each such branch ends in a *node*, which can be thought of as first level nodes or depth one nodes. The branches are *labeled* by a -1- or -0- (for a binary tree). By convention, we often put the label -1- on the upper branch in a horizontally drawn tree and a -0-

on the lower branch. Nodes either have further branches leading to more nodes, or they are *terminal nodes* or *leaves* with no extending branches. This tree is depicted as growing from left to right, but they are often drawn in vertical fashion with the root on the bottom (like most biological trees) or with the root on top and the branches extending downward. A level $n+1$ node connected by a branch from a level n node is said to be a *child* of the latter node, which is called the *parent* of the level $n+1$ node. Children of a common parent are called *siblings*. There is a one-to-one correspondance between paths from the root node to the leaves and the codewords. The codewords are for this reason sometimes called "path maps". Reading the branch labels from the root on the left to the leaf on the right yields a binary codeword. By construction of the tree, no codeword can be a prefix of another codeword since codewords terminate in leaves, i.e., no other codewords begin with the same binary sequence. Conversely, given any prefix code we can represent as a tree. An encoder is a means of assigning one of the codewords to a source symbol. It might (or might not) take adventage of the tree structure.

The Kraft Inequality

A necessary conditio for unique decodability of a noiseless source code with input alphabet $A = \{ a_0, \ldots , a_{M-1} \}$, encoder α, and codeword lengths $l_k = l (a_k)$, $k = 0,1,\ldots, M-1$, is

$$\sum_{k=0}^{M-1} 2^{-l_k} \leq 1.$$

Binary codewords of length l_{M-1} and shorter can be considered as paths through the tree or, equivalently, as the terminal nodes of such a path. In the *figure 3* a complete tree is depicted with each branch being labeled by a 0 or 1. The code is represented by the *subtree* consisting

of the branches from the root of the tree to the terminal nodes (leaves) of the subtree denoted by the circles. The codewords correspond to the sequences of the branch labels from the root of the tree to the leaf. The lengths of the codewords in the figure are
{ 1,2,3,4,4 }. The codewords corresponding to the leaves of the subtree are given in the boxes near the leaves.

In a general binary tree of arbitrary depth, a codeword of length l correspods to a path of l branches in the tree beginning at the root node (depth 0) and finishing at a terminal node of depth l in the tree. The codeword is the sequence of binary labels of the branches read from the first branch to the branch at depth l. Given a collection of lengths satisfy the Kraft inequality, pick an arbitrary node of depth $\boldsymbol{l}_0$ and hence an arbitrary length l_0 binary sequence as the first codeword. In *figure 3* this first choice is the single symbol sequence 0 corresponding to the downward branch emanating from the root node. Since no other codeword can have this first codeword as prefix, we prune the tree at the terminal node of this first codeword at depth $\boldsymbol{l}_0$ in the tree. This removes all of the deeper nodes emanating from the terminal node of the first codeword from consideration as terminal nodes for the other codewords.

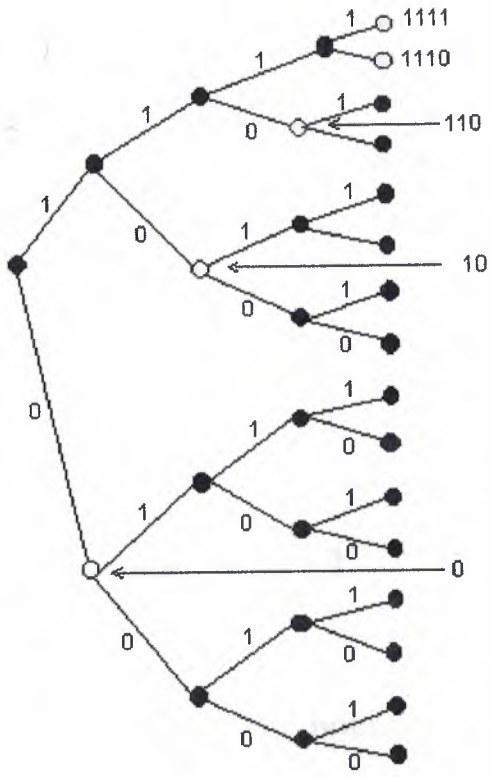

figure 3

Next pick one of the remaining available depth l_1 nodes and hence the corresponding binary l_1 -tuple as the second codeword. In *figure 3* this is the length 2 sequence 10. Observe that there are $2^{l_1} - 2^{l_1 - l_0}$ available nodes at this depth.

The Kraft inequality proides the basis for simple lower and upper bounds to the average length of inequaly decorable variable length noiseless codes. The remainder of this section is devoted to the development of the bound and some of its properties.

Entropy

We have from the Kraft inequality that

$$l(\alpha) = \sum_{a \varepsilon A} p(a)\, l(a)$$

$$= - \sum_{a \varepsilon A} p(a) \log 2^{-l(a)}$$

$$\geq - \sum p(a) \log \left(2^{-l(a)} / \sum_{b \varepsilon A} 2^{-l(b)} \right),$$

formula 3

where the logarithm is base 2. The bound on the right-hand side has the form

$$\sum p(a) \log (1 / q(a))$$

for two pmf's p and q. the following lemma provides a basic lower bound for such sums that depends only on p.

Let us now consider the *divergence inequality*:

Given any two pmf's p and q with a common alphabet A, then

$$D(p||q) \equiv \sum p(a) \log (1 / q(a)) \geq H(p) \equiv \sum p(a) \log (1 / p(a))$$

formula 4

$D(p||q)$ is called the *divergence inequality or* <u>*relative entropy*</u> *or cross entropy* of the pmf's p and q. $H(p)$ is called the *entropy* of the pmf p or, equivalently, the entropy of the random variable X described

NEAR EAST UNIVERSITY

1988

COMPUTER ENGINEERING DEPARTMENT

GRODUATION PROJECT COM 400 ENTROPY CODING

SUPERVISOR :
FAHRETTIN M. SADIGOGLU

MUSTAFA DINC
92304

MUTLU SAYAR
93078

JUNE 1997

TABLE OF CONTENTS

INTRODUCTION

First of all we should deal with what a source coding is. In *figure1* it is shown a schematic diagram of a transmission system. It contents most of the components of a transmission system.

A trnsmission system transmits signals from a source to a destination, which sometimes called a sink. The source is the device by which the signals are generated. The destination is the place where they are received.

The central part of the transmissin system is the channel. This is the medium which physically transports the signal, for example electrical sygnal, as an electromagnetic wave or as a sequence of light pulses. Practical examples of channels are a coaxial cable, an optical fiber and the atmosphere. Apart from the channel, the sygnal passes through three pairs of blocks: the modulator and the demodulator blocks, the error protection and correction blocks, and the source encoder and decoder blocks. These pairs of blocks are present because of three channel properties discused below.

The first channel property is that channels can only transport physical (e.g. electrical) signals. For successful transmision a digital sygnal must therefore be converted to a form appropriate for the channel. For radio transmission, for instance, the signal must be modulated on a carrier and then converted into an electromagnetic wave. The modulator converts the signal to a representation that is appropriate for the channel. The demodulator converts the received signal back to its original form.

The second channel property is that, even if a signal is adapted to the channel, it does not always pass it undisturbed: the channel may introduce errors. If digital information is transmitted, the result can be that after demodulation some bits are inverted. The block before the modulator is the error-protection unit. It adds bits to the signal that

make it possible for the error-correction unit to detect and even correct errors introduced by the channel.

The third channel property is that there is an upper bound to the number of bits per second that can be correctly transmitted. This bound is called the channel capacity. The source-encoding block reduces the number of bits per second with which the input signal is represented, to a number that is low enough for transmission. The signal with the reduced bit rate is the source-encoding signal. The source decoder converts this to a reconstruction of the input signal. Unfortunately, source encoding and decoding may change the signal. This results in the reception of a distorted signal. In a good source-coding system the distortion is kept below a certain level.

It is mainly source coding for speech, music and pictures that is considered here. This implies that one finds a human observer at the destination. This has its impact on the notion of distortion and on the design of source-coding systems. Agood source-coding system keeps distortion below a certain level. If signals such as speech, music and pictures are received by a human observer, it means that after reception these signals must have a desired subjective quality rather than a desired objective quality.

In most of what follows it is assumed that the concentrantion of the error-protection block, the modulator, the channel, the demodulator and the error-correction block behaves as a digital, error-free channel, which imlies that the source decoder receives the undestorted output oh the source encoder.

Source coding is not only the name for the discipline involved with the design of source-coding algorithms and systems but also for the action of the source encoder and decoder. Other names that are sometimes used for the source coding are bit-rate reduction, data reduction and data compression. The combination of a source encoder and decoder is often called codec.

Examles of source coding applied in transmission and storage systems are: source codig of speech signals in mobile automatictelephony, source coding of x-ray and nuclear magnetic resonance images for storage in medical databases, source coding of sound signals for storage on compact disc interactive (CD-I) disks and on digital compact cassette (DCC) tapes and for digital audio broadcasting, source coding of images of documents for storage in the Megadoc system, and source coding of digital TV pictures for storage on a digital video tape.

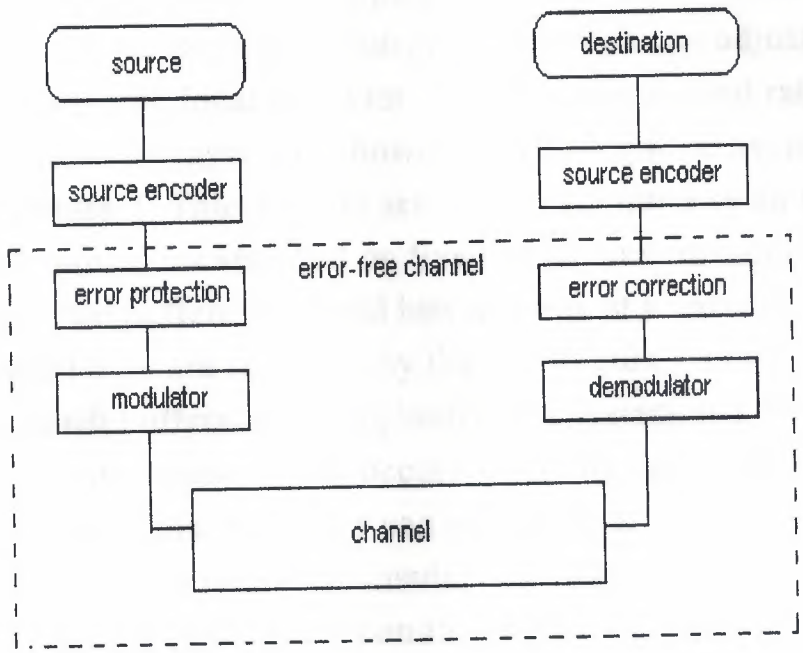

Figure1

ENTROPY CODING

Most of the coding systems are *fixed rate codes* in the sense that a fixed number of channel bits per time unit is produced by the encoder and processed by decoder. Examples of these type of codes are quantization, bit allocation and transform coding. In some communication and storage systems, fixed rate operation is not desirable because the data source may display wide variations of activity. For example, samled speech may change very little during long periods of silence and then exhibit very complex behavior during plosives. Ideally, one would like to waste few bits coding the silence and preserve them for coding the highly informative transitients. Such a strategy requires a *variable rate code*, a code which can adjust its own bit rate to better match local behavior. In order to use fixed rate communication and storage links, however, the long term average bit rate must be constant. Thus buffers are usually required as an interface when variable rate codes are used on fixed rate communication or storage media. The buffers will hold bits arriving at a variable rate from the encoder until they are accepted by the fixed rate channel for transmission. Such buffers add complexity to a system and can also add errors when they overflow, which occurs when the data source produces bits faster than the buffer can accept them. Similarly, errors can be introduced when the buffers underflow, which occurs when the data source produces bits slower than the rate at which the buffer is releasing bits. To combat this problem, a technique known as *buffer feedback* is commonly used, where the occupancy level of the buffer is fed back to the source encoder to suitably adjust the quantizer data rate. This added complexity is often justified, however, by the potentially significant performance gains possible with the variable rate strategies. Entropy codes are often used in conjunction with scalar quantizers (to conserve the average bit rate) and are often fairly simple to implement when the input alphabets are of reasonable size.

The overall variable rate code is then a simple cascade of a scalar quantizer, which performs the analog-to-digital conversion in a fixed rate manner, and a variable length noiseless code, which maps the quantizer output into a variable length binary index in a way that can be perfectly decoded by the receiver.

Communication and storage systems that are inherently variable rate are increasing in importance and variable length codes can be well matched to such systems. For example, variable rate codes cause no problems in offline starage (the bits are accepted as they come until the file is complete) and variable rate codes are no more complicated than fixed rate codes for use in packet communication environments.

Entropy coding is also often referred to as *noiseless coding, lossless coding,* and *data compaction coding.* It is also referred to simply as *data compression* in the computer science literature, but it is avoided that this nomenclature as entropy coding is a very special case of data compression. The narrow use of the term by computer scientists is perhaps understandable because of the disastrous consequences that can result from even rare bit errors if the compressed file is a binary executable file. When bit errors cause catastrophe, lossy codes are not useful for compression (except possibly as a component of an overall lossless code).

The goal of noiseless coding is to reduce the average number of symbols sent while suffering no loss of fidelity. A classical example is the Morse code where short binary codewords are used for more probable letters and long codewords used for less probable letters. The Morse code in fact is a very good code for its age and, when applied to English text, results in many fewer bits on the average than would the use of one byte ASCII codes for each letter. A more recent but still venerable example is the run-length code used to code sources which tend to repeat symbols for long periods of time. For example, a binary source such as facsimile may produce long runs of zeros and occasionally, ones. Hence one means of compression is to sequentially

send a symbol followed by the number of its repetitions, the *run lenth.* This will result in compression on the average if the source tends to produce such runs. It will not compress a memoryless source.

Variable-Length Scalar Noiseless Coding

note: In my report I tried to avoid using mathematical expressions but I used the ones that are unavoidable for explaining the event.

Suppose that $\{ X_n \}$ is a stationary sequence of random variables with a finite alphabet $A = \{ a_0, \ldots, a_{M-1} \}$ with a marginal probability mass function $p(a) = p_x (a) = \Pr (X_n = a)$. The case of of primary interest for the present purposes is that where the X_n are quantized versions of continuous alphabet sequence W_n , that is, $X_n = Q (W_n)$, with q an ordinary scalar quantizer.

A *variable length scalar noiseless code* consists of an encoder α , which maps a single input symbol χ in A into a binary vector $\alpha (\chi)$ of dimension or length $l(\chi)$, and a decoder β , which maps binary vectors u of differing length into an output $\beta (u)$ so that $\beta (\alpha (\chi)) = \chi$; that is, the encoding / decoding operation is *lossless* or *noiseless* or *transparent*. The goal of the code is to keep the average number of bits transmitted for each source symbol as small as possible, that is, to minimize the *average length*

$$l (\alpha) = E\, l (X_n) = \sum_{A \varepsilon A} p (a)\, l (a).$$

formula 1.

If *formula 1* is accepted as a definition of quality of noiseless source code, then it is of interest to quantify how small $l (\alpha)$ can be made and hence what the optimal achievable performance is. It is also

of interest to construct actual codes that perform very near to the optimal quantity.

Unfortunately, the given definition of a code is not enough to ensure that it is useful. Suppose, for example, that the input alphabet has 4 letters,

$A = \{ a_0, a_1, a_2, a_3 \}$, possibly the output of a 2 bit per sample quantizer.

Input letter	Codeword
a_0	0
a_1	10
a_2	101
a_3	0101

table 1.

Although this is a noiseless code by the above definition, it cannot always be decoded in a noiseless fashion when the code is applied to a sequence of inputs. For example, if the receiver gets the sequence 0101101.... , it could have been produced by the input sequence $a_0a_2a_2a_0a_1$.... or by $a_3a_2a_0a_1$... . To make matters worse, the ambiguity can never be resolved regardless of future received bits. Hence for a code to be useful, it must be *uniquely decodable* in the sence that if the decoder receives a valid encoded sequence of finite length, there is only one possible input sequence that could have produce the encoded sequence. The effectively extends the idea of a noiseless or transparents code from a single letter to a sequence. Note that we could accomlish this by inserting punctuation in the binary sequence between codewords, e.g., add a third letter "," and send the sequence 0, 101, 101, 0, 10,... While this disambiguates the sequence, it also increqses the average length of the encoding as well as the required channel alphabet. This may be a simple fix, but it is not an efficient use of symbols. An alternative and less restrictive approach is to require that the code satisfy a *prefix condition* in the sense that no codeword be a prefix of any other codeword. In the previous example, a_0 is a prefix of

a_3 and a_1 a prefix of a_2. An example of a code satisfying the prefix condition is given below.

Input letter	Codeword
a_0	0
a_1	10
a_2	110
a_3	111

table 2.

Binary prefix codes can be depicted as a binary *tree* as below.

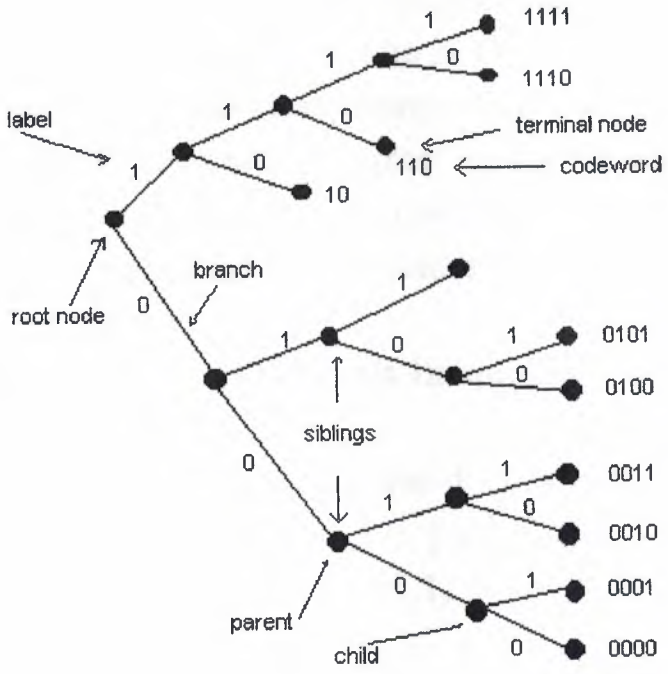

figure 2.

The binary tree starts with a *root* or *root node* which has *branches* extending from it. Each such branch ends in a *node*, which can be thought of as first level nodes or depth one nodes. The branches are *labeled* by a -1- or -0- (for a binary tree). By convention, we often put the label -1- on the upper branch in a horizontally drawn tree and a -0-

on the lower branch. Nodes either have further branches leading to more nodes, or they are *terminal nodes* or *leaves* with no extending branches. This tree is depicted as growing from left to right, but they are often drawn in vertical fashion with the root on the bottom (like most biological trees) or with the root on top and the branches extending downward. A level $n+1$ node connected by a branch from a level n node is said to be a *child* of the latter node, which is called the *parent* of the level $n+1$ node. Children of a common parent are called *siblings*. There is a one-to-one correspondance between paths from the root node to the leaves and the codewords. The codewords are for this reason sometimes called "path maps". Reading the branch labels from the root on the left to the leaf on the right yields a binary codeword. By construction of the tree, no codeword can be a prefix of another codeword since codewords terminate in leaves, i.e., no other codewords begin with the same binary sequence. Conversely, given any prefix code we can represent as a tree. An encoder is a means of assigning one of the codewords to a source symbol. It might (or might not) take adventage of the tree structure.

The Kraft Inequality

A necessary conditio for unique decodability of a noiseless source code with input alphabet $A = \{ a_0, \ldots , a_{M-1} \}$, encoder α, and codeword lengths $l_k = l (a_k)$, $k = 0,1,\ldots, M-1$, is

$$\sum_{k=0}^{M-1} 2^{-l_k} \leq 1.$$

Binary codewords of length l_{M-1} and shorter can be considered as paths through the tree or, equivalently, as the terminal nodes of such a path. In the *figure 3* a complete tree is depicted with each branch being labeled by a 0 or 1. The code is represented by the *subtree* consisting

of the branches from the root of the tree to the terminal nodes (leaves) of the subtree denoted by the circles. The codewords correspond to the sequences of the branch labels from the root of the tree to the leaf. The lengths of the codewords in the figure are
{ 1,2,3,4,4 }. The codewords corresponding to the leaves of the subtree are given in the boxes near the leaves.

In a general binary tree of arbitrary depth, a codeword of length l correspods to a path of l branches in the tree beginning at the root node (depth 0) and finishing at a terminal node of depth l in the tree. The codeword is the sequence of binary labels of the branches read from the first branch to the branch at depth l. Given a collection of lengths satisfy the Kraft inequality, pick an arbitrary node of depth $\boldsymbol{l}_0$ and hence an arbitrary length l_0 binary sequence as the first codeword. In *figure 3* this first choice is the single symbol sequence 0 corresponding to the downward branch emanating from the root node. Since no other codeword can have this first codeword as prefix, we prune the tree at the terminal node of this first codeword at depth $\boldsymbol{l}_0$ in the tree. This removes all of the deeper nodes emanating from the terminal node of the first codeword from consideration as terminal nodes for the other codewords.

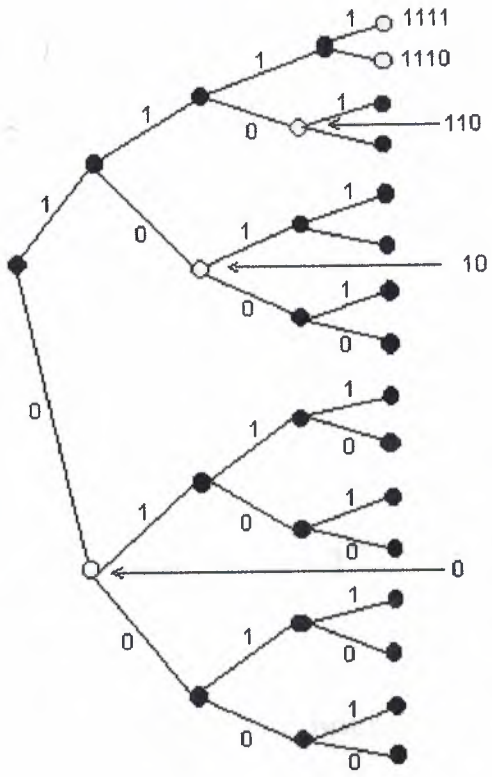

figure 3

Next pick one of the remaining available depth l_1 nodes and hence the corresponding binary l_1 -tuple as the second codeword. In *figure 3* this is the length 2 sequence 10. Observe that there are $2^{l_1} - 2^{l_1 - l_0}$ available nodes at this depth.

The Kraft inequality proides the basis for simple lower and upper bounds to the average length of inequaly decorable variable length noiseless codes. The remainder of this section is devoted to the development of the bound and some of its properties.

Entropy

We have from the Kraft inequality that

$$l(\alpha) = \sum_{a \varepsilon A} p(a)\, l(a)$$

$$= - \sum_{a \varepsilon A} p(a) \log 2^{-l(a)}$$

$$\geq - \sum p(a) \log \left(2^{-l(a)} / \sum_{b \varepsilon A} 2^{-l(b)} \right),$$

formula 3

where the logarithm is base 2. The bound on the right-hand side has the form

$$\sum p(a) \log (1 / q(a))$$

for two pmf's p and q. the following lemma provides a basic lower bound for such sums that depends only on p.

Let us now consider the *divergence inequality*:

Given any two pmf's p and q with a common alphabet A, then

$$D(p||q) \equiv \sum p(a) \log (1 / q(a)) \geq H(p) \equiv \sum p(a) \log (1 / p(a))$$

formula 4

$D(p||q)$ is called the *divergence inequality or* <u>*relative entropy*</u> *or cross entropy* of the pmf's p and q. $H(p)$ is called the *entropy* of the pmf p or, equivalently, the entropy of the random variable X described

by the pmf p. Both notations $H(p)$ and $H(X)$ are common, depending on whether the emphasis is on the distribution or on the random variable.

Divergence inequality immediately yields the following lower bound:

Given a uniquely decodable scalar noiseless variable length code with encoder α operating on a source X_n with marginal pmf p, then the resulting average codeword length satisfies

$$l(\alpha) \geq H(p);$$

formula 5

that is, the average length of the code can be no smaller than the entropy of the marginal pmf. The inequality is an equality if and only if

$$p(a) = 2^{-l(a)} \qquad \text{for all } a \varepsilon A.$$

formula 6

Note that the equality in *formula 6* follows when both *formula 3* and *formula 4* hold with equality. The latter equality implies that $q(b) = 2^{-l(b)}$.

Because the entropy provides a lower bound to the average length of noiseless codes and because, as we shall see, good codes can perform near this bound, uniquely decodable variable length noiseless codes are often called *entropy codes*. To achieve the lower bound, we need to have *formula 6* satisfied. Obviously, however, this can only hold in the special case that the input symbols all have probabilities that are powers of 1/2 . In general $p(a)$ will not have this form and hence the bound will not be exactly achievable. The practical design goal in this case is to come as close as possible.

Prefix Codes

Prefix codes were introduced under title -variable length scalar noiseless coding- as a special case of uniquely decodable codes wherein no codeword is a prefix of any other code word. Assuming a known staring point, decoding a prefix code simply involves scanning symbols until one sees a valid codeword. Since the codeword cannot be a prefix for another codeword, it can be immediately decoded. Thus each codeword can be decoded as soon as it is complete, without having to wait for further codewords to resolve ambiguitites. Because of this property, prefix codes are also referred to as *instantaneous codes*.

Although prefix codes appear to be a very special case, the following theorem demonstrates that prefix codes can perform just as well as the best uniquely decodable code and hence no optimality is lost by assuming the prefix code must satisfy the Kraft inequality and hence there must exist a prefix code with these lengths.

Suppose that (α, β) is a uniquely decodable variable length noiseless source code and that $\{ l_m ; m = 0,1,\ldots\ldots, M\text{-}1 \} = \{ l (a); a \in A \}$ is the the collection of codeword lengths. *Then there is a preifx code with the same lengths and the same average length.*

The theorem implies that the optimal prefix code, wher here *optimal* means providing the minimum average length, is as good as the optimal uniquely decodable code. Thus we lose no generality by focusing henceforth on the properties of optimal prefix codes. The following theorem collects the two most important properties of optimal prefix codes.

An optimum binary prefix code has the following properties:

- *if the codeword for input symbol a has length l(a), then p(a) > p(b); that is, more probable input symbols have shorter (at least, not longer) codewords.*

- *the two least probable input symbols have codewords which are equal in length and differ only in the final symbol.*

Let us prove these properties:

If $p(a) > p(b)$ and $l(a) > l(b)$, then exchanging codewords will cause a strict decrease in the average length. Hence the original code could not have been optimum.

Suppose that the two codewords have different lengths. Since a prefix of the longer codeword cannot itself be a codeword, we can delete the final symbol of the longer codeword without truncated word being confused for any other codeword. This strictly decreases the average length of the code and hence the original code could not have been optimum. Thus the two least probable codewords must have equal length. Suppose that these two codewords differ in some position other than the final one. If this were true, we could remove the final binary symbol and shorten the code without confusion. This is true since we could still distinguish the shorter codewords and since the prefix condition precludes the possibility of confusion with another codeword. This, however, would yield a strict decrease in aveage length and hence the original code could not have been optimum.

The theorem provides an iterative design technique for optimal codes, as will be in the next section.

Huffman Coding

In 1952 D.A. huffman developed a scheme which yields performance quite close to the lower bound of suuficiency theorem. In fact, if the input probabilities are powers of 1/2, the bound is achieved. The design is based on the ideas of the second theorem of prefix codes. Suppose that we order the input symbols in terms probability, that is, $p(a_0) \geq p(a_1) \geq \ldots\ldots \geq p(a_{M-1})$. Depict the symbols and probabilities as a list $\mathfrak{I}$ as in *table 3*.

Symbol	probaility
a_0	$p(a_0)$
a_1	$p(a_1)$
:	:
:	:
a_{M-2}	$p(a_{M-2})$
a_{M-1}	$p(a_{M-1})$

table 3: list $\mathfrak{I}$

The input alphabet symbols can be considered to correspond to the terminal nodes of a code tree which we are to design. We design this tree from the leaves back to the root stages. Once completed, the codewords can be read off from the tree by reading the sequence of branch labels encountered passing from the root to the leaf corresponding to the input symbol.

The theorem implies that the two least probable symbols have codewords of the same length which differ only in the final binary symbol. thus we begin a code tree with two terminal nodes with branches extending back to a common node. Label one branch 0 and the other 1. We now consider these two input symbols to be *tied* together and form a single new symbol in a reduced alphabet A' with

M-1 symbols in it. Alternatively, we can consider the two symbols a_{M-2} and a_{M-1} to be merged into a new symbol (a_{M-2}, a_{M-1}) having as probability the sum of the probabilities of the original nodes. We remove these two symbols from the list $\mathfrak{I}$ and add the new merged symbol to the list. This yields the modified list of *table 4*.

We next try to find an optimal code for the reduced alphabet A' (or modified list $\mathfrak{I}$) with probabilities $p(a_m)$; $m = 0, 1, \ldots, M-3$ and $p(a_{M-1}) + p(a_{M-2})$. A prefix code for A by adjoining the final branch labels already selected. Furthermore, if the prefix code for A' is optimal, then so is the induced code for A. to prove this, observe that the lengths of the codewords for a_m; $m = 0,1,\ldots,M-3$ in the two codebooks are the same. The codebook for A' has a single word of length l_{M-2} for the combined symbol (a_{M-2}, a_{M-1}) while the codebook for A has two words of length $l_{M-2} + 1$ for the two input symbols a_{M-2} and a_{M-1}. This means that the average length of the codebook for A is that for the codebook for A' plus $p(a_{M-1})$, a term which does not depend on either codebook. Thus minimizing the average length of the code for A' also minimizes the average length of the induced code for A.

symbol	probability
a_0	$p(a_0)$
a_1	$p(a_1)$
:	:
:	:
(a_{M-1}, a_{M-2})	$p(a_{M-1})+p(a_{M-2})$

table 4: list $\mathfrak{I}$ after one Huffman step

We continue in this fashion. The probability of each node is found by adding up the probabilities of all input symbols connected to

the node. At each step the two least probable nodes in the tree are found. Equivalently, the two least probable symbols in the ordered list $\mathfrak{I}$ are found. These nodes are tied together and a new node is added with branches to each of the two low probability nodes and with one branch labeled 0 and the other 1. The procedure is continued until only a single node remains (the list contains a single entry). The algorithm can be summerized in a concise form due to Gallager as in *table 5*.

Huffman Code Design
1. Let $\mathfrak{I}$ be a list of the probabilities of the source letters which are considered to be associated with the leaves of a binary tree. 2. Take the two smallest probabilities in $\mathfrak{I}$ and make the corresponding nodes siblings. Generate an intermediate node as their parent and label the branch from parent to the other child 0. 3. Replace the two probabilities and associated nodes in the list by the single new intermediate node with the some of the two probabilities. If the list now contains only one element, quit. Otherwise go to step 2.

Table 5

An example of the construction is depicted in *figure 4*.

Observe that a prefix code tree combined with a probability assignment to each leaf implies a probability assignment for every node in the tree. The probabilities of two children sum to form the probability of their parent. The probability of root node is 1.

We have demonstrated that the above technique of constructing a binary variable length prefix noiseless code is optimal in the sense that

no binary variable length uniquely decodable scalar code can give a strictly smaller average length. Smaller average length could, however, be achieved by relaxing these conditions. First, one could use nonbinary alphabets for the codebooks, e.g., ternary or quaternary. Similar constructions exist in this case. Second, one could remove the scalar constraint and code successive pairs or larger blocks or vectors of input symbols, that is, consider the input alphabet to eb vectors of input symbols instead of only single symbols.

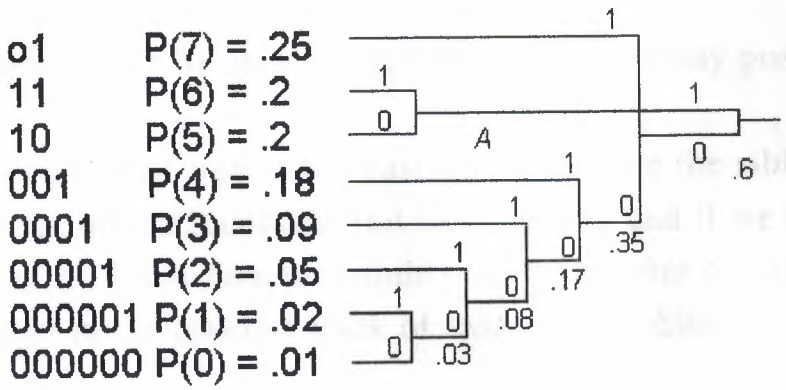

figure 4: a Huffman code

The Sibling Property

In this section we describe a structural property of Huffman codes due to Gallager. This provides an alternative characterization of Huffman codes and is useful in developing the adaptive Huffman code to be seen later.

A binary code tree is said to have the *sibling property* if

1. every node in the tree (except for the root node) has a sibling.
2. the nodes can be listed in order of decreasing propability with each proabilitites.

The list need to be unique since distinct nodes may posses equal probabilities.

The code tree of *figure 4* is easily seen to have the sibling property. Every node except the root has a sibling and if we list the nodes in order of decreasing probability we have *table 6*. Each successive pair in the ordered stack of *table 6* is a sibling pair.

.6
.4
.35
.25
.2
.2
.18
.17
.09
.08
.05
.03
.02
.01

table 6

A binary prefix code is a Huffman code if and only if it has the sibling property.

First, assume that we have a binary prefix code design algorithm as follows:

1. Let $\mathfrak{I}$ be a list of the probabilities of the source letters which are considered to be associated with the leaves of a binary tree. Let ω be be a list of nodes of the code tree; initially ω is empty.
2. Take two of the smallest properties in $\mathfrak{I}$ and make the corresponding nodes siblings. Generate an intermediate node as their parent and label the branch from parent to one of the child nodes 1 and label the branch from the parent to the other child 0.
3. Replace thw two probabilities and associated nodes in the list $\mathfrak{I}$ by the new intermediate node with the sum of the two probabilities. Add the two sibling nodes to the top of the list ω, with the higher probability node on top. If the list $\mathfrak{I}$ now contains only one element, wuit. Otherwise goto step 2.

The list ω is constructed by adding siblings together, hence siblings in the final list are always adjacent. The new additions to ω are chosen from the old $\mathfrak{I}$ of step 2 by choosing the smallest probability nodes. Thus the two new additions have smaller probabilities (at least no greater) than all the remaining nodes in the old $\mathfrak{I}$. This in turn imlies that these new additions have smaller probabilities than all of the nodes in the new $\mathfrak{I}$ formed by merging these two nodes in the old $\mathfrak{I}$. Thus in the next iteration the next siblings to be added to the list ω must have probability no smaller than the current two siblings since the next ones will be choosen from the new $\mathfrak{I}$. Thus ω has adjacent siblings listed in the order of descending probability and therefore the code has the sibling property.

Next suppose that a code tree has the sibling property and that ω is the corresponding list of nodes. The bootm (smallest probability) nodes in this list are therefore siblings. Suppose that one of these nodes is an intermediate node. It must have at least one child which in turn must have a sibling from the sibling property. As the probabilities of the siblings must sum to that of the parent, this means that the children must have smaller probability than the parent, which contradicts the assumption that the parent was one of the lowest probability nodes. (This contradiction assumes that all the nodes have nonzero probability, which we assume without any genuine loss of generality). Thus these two bottom nodes must in fact be leaves of the code tree and they correspond to the lowest probability source letters. Thus the Huffman algorithm in this first pass will in step 2 assign siblings in the original code tree to these two lowest probability symbols and it can label the siblings in the same waythat the siblings are labeled in the original tree. Next remove these two siblings from the code tree and remove the corresponding bottom two elements in the ordered list. The reduced code tree still has the sibling property and corresponds to the reduced code tree $\mathfrak{I}$ after a complete pass through the Huffman algorithm. This argument can be applied again to the reduced lists: At each pass the Huffman algorithm chooses as siblings from the original prefix code tree and labels the corresponding branches exactly as the original siblings were labeled in the original tree. Continuing in this manner shows that the Huffman algorithm "guided" by the original tree will reduce the same tree.

Vector Entropy Coding

All of the entropy coding results developed thus far apply immediately to the "extended source" consisting of successive nonoverlapping N-tuples of the original source . in this case the entropy lower bound becomes the enropy of the input vectors instead of the marginal entropy. For example, if px^N is the pmf for a source vector $X^N = (X_0, \ldots\ldots, X_{N-1})$, then a uniquely decodable noiseless source code for successive source blocks of length N has an average codeword length no samller than

$$H(px^N) = \Sigma - px^N (\chi^N) \log px^N (\chi^N),$$

the Nth order entropy of the input. This is often written in terms of the average codeword length per input symbol and entropy per unit symbol:

$$l \geq (1 /N) H(X^N).$$

formula 7

This bound is true for all N.

For any integer N a prefix code has averasge length satisfying the lower bound of formula 7. Furthermore, there exist a prefix code for which

$$l < (1 / N) H(X^N) + 1 / N.$$

formula 8

It can be shown that if the input process is stationary, then one can take the minimum over N on the right hand side to achieve lower bound for

all N and that this minimum is $H^{\wedge}$, the *entropy rate* of the source defined by

$$H^{\wedge} = \lim (H(XN) / N).$$

The construction of Huffman codes extends in principle to such block inputs, but obviously the technique becomes far more complicated as the number of input symbols grows. Furthermore, if we are going to code groups of input symbols into groups of code symbols, then the previous approach of coding fixed length input blocks into variable length output blocks is not the only possible code structure. While a Huffman code may be optimal for this structure, other code structures may provide superior performance, that is, smaller average length with comparable or less complexity. One can consider codes that map variable length blocks into fixed length blocks and codes that map variable length blocks into variable length blocks. We next turn the alternative noiseless coding techniques.

Atithmetic Coding

Arithmetic coding is a direct descendentof an unpublished coding technique of P. Elias that was developed by Pasco and Rissanen and subsequently improved by Rissanen and Langdon, jones and others. A good tutorial overview and reference list may be found in the paper Witten et al.

We demonstrate the basic idea by focusing on an example of the Elias code itself. Arithmetic codes can be viewed as Elias codes with finite precision arithmetic, that is, codes which do not assume arbitrary arithmetic precision. To simplify the description we also restrict interest to a memoryless binary source. Extensions involve similar ideas with added complexity. Since we wish to compress a source with only two symbols, clearly we will have to code groups of input symbols.

Unlike the vector entropy coders, however, now the number of input symbols grouped together will vary.

Once again the code will be described by a tree, a binary tree for the case of designing a code for a binary source. A good way to think the structure of the tree is as a *classification tree* for points in the units interval

[0,1). The tree will have as an input a real number $r \in [\ 0,1\ )$ and each node will make a binary decision based on r. based on this decision the classifier will either output a 1 and advance along the (say) right branch emenating from the node or output a 0 and advance along the left. The tree will not much resemble its eventual application of noiseless code while we construct it, instead it will look like a means of assigning a binary sequence to a real number, reminiscent of the ordinary binary expansion of numbers in the unit interval:

$$r = \sum_{i=1}^{\infty} u_i \, 2^{-i},$$

formula 9

where u_i are 0 or 1. This expansion will play a key role in using the tree as a code, but the tree itself will not try to produce a binary sequence { u_i } for a real number r for which *formula 9* is true, instead it will try to trace a path through the tree for a given "input" r with the following property: If r is selected at random according to a uniform distribution, then the probability of having the classifier produce a given binary k-tuple after k decisions is the same as the probability of the original binary source producing that binary

k-tuple. The tree can be thought of as a *model* for the source, a means for producing binary sequences having the same probabilities as the original source by making a sequence of deterministic decisions on a

uniform random variable. Stated in another way, there is a one-to-one correspondence between source binary k-tuples and tree path maps from the root to depth k. We show how such a classifier / madel can be constructed and then demonstrate how the resulting tree can be used to noiselessly encode the original source.

Suppose that the input is a memoryless binary source with alphabet { 0,1 } and *pmf* $p(0) = q$, $p(1) = 1\text{-}q$ and entropy

$$H(p) = - q \log q - (1\text{-}q) \log (1\text{-}q).$$

consider the unit interval [0,1) to consist of two subintervals with length proportional to the two input letter probabilities: [0,q) and [q,1). We can then subdivide each of these intervals into sub intervals with length proportional to the two letter probabilities. Now the four subintervals have lengths corresponding to the probabilities of all 2 dimensional source blocks:

$$q^2 \ , \ q(1\text{-}q) \ , \ (1\text{-}q)q \ , \ (1\text{-}q)^2.$$

From the modeling standpoint, a uniform input to this two-level tree will produce four binary pairs with the same probaility as will the original source. There is a one-to-one correspondence between pairs (binary two-tuples) from the source and these four subintervals having length equal to the probabilities of all possible binary two-tuples. Hence we could "code" the input pairs into subintervals in an invertible manner; that is, we could assign a subinterval to each binary pair and infer without error the original pair from the subinterval.

We continue this idea recursively subdivide the unit interval into smaller subintervals so that after the nth subdivision there would be 2^n subintervals with lengths equal to the probabilities of all possible $2n$ binary
n-tuples.

As with pairs, in principal we could assign to each input k-tuple the one fo the possible $2k$ intervals having as length the probability of of the
k-tuple. This coding is invertible, but it is not yet a practical noiseless coding scheme because the end points of the intervals are in general real numbers; it is not clear how to convert these :codewords" into binary codewords for communication purposes.

A this point recall the binary expansion of *formula9* for $r \in [\,0,1\,]$. The endpoints of any of the intervals can be expressed in such a fashion. This clearly assigns at least an infinite binary sequence to each interval, but in fact we can find a finite binary sequence which will tell us if we are in a particular interval or not. This finite binary sequence will serve as the codeword that specifies the interval and therefore the original source binary sequence. The idea is equivalent to viewing the source sequence as a specification of the random number r with increasing resolution as successive source symbols arrive. For a given number of source digits the encoder generates a codeword that represents the convetional binary expansion of that number r to the best precision possible from the available number of source digits.

Suppose we wish to encode a binary sequence $x_0, x_1, \ldots$, first look at x_0 and the corresponding interval I_0 in [0,1). If the both ends of I_0 have the same first term in the binary expansion (which means the interval is entirely in [0, ½) or [½, 1), then the encoder will release that common bit. This becomes the first bit of the output codeword. If the first two binary symbols of the binary expansion of the endpoints of the interval corresponding to x_0 agree, then the encoder will check the next binary symbol. as many such symbols that agree are released to the channel. When no more binary symbols match, the encoder proceeds to the next input symbol x_1 and inspects its corresponding interval. If the decoder gets a bit at time 0, it eill know which of the intervals was seen and hence what x_0 was.

If, on the other hand, the interval endpoints of the first interval do not have a common first binary symbol, the encoder sends nothing and instead immediately inspects the second input symbol x_1. The observed pair x_0, x_1 now corresponds to a specific subinterval of $[0,1)$ with length equal to the probability of x_0, x_1. The encoder again checks the endpoints of this interval to see if the first binary term agrees (which, as berfore, would specify that subinterval lat either entirely on the left or entirely on the right of ½). If the first binary symbol agrees, the encoder can release that binary symbol to the decoder. On seeing that symboln the decoder will be able to determine x_0 as above. The encoder can then check the second binary symbol in the expansion for the endpoints of the second level subinterval. If they agree, the common symbol is released. If not, a new symbol is checked.

If even the first binary symbols of the endpoints do not agree, then the encoder must look at more input symbols before releasing any binary symbols.

The encoder continues in this way: at each time it views a new input symbol and then looks at the endpoints of the corresponding subinterval. The subintervals are shrinking with each new input symbol. If there are any new binary symbols in accord (symbols not already sent), then they are released to the decoder. As the decoder receives these binary symbols, it can determine with any increasing accuracy a subinterval (having length a power of ½) which contains the "code" subinterval and hence can reconstruct the input sequence.

Consider only the effects of coding the first three source symbols, x_0, x_1, x_2. *Table 7* shows the source symbols, the resulting binary words produced by the Elias code after the three source symbols have been encoded, the resulting length of the codeword, and the probability of seeing that source sequence and hence having that length.

x_0, x_1, x_2	$u_0, u_1, u_2, u_3, u_4, u_5$	l	$p(x0,x1,x2)$
111	1	1	27/64
110		0	9/64
101	01	2	9/94
100	0100	4	3/64
011	00	2	9/64
010	0001	4	3/64
001	0000	4	3/64
000	000000	6	1/64

table 7

The average channel codeword length considering only the encoder up to three symbols is

$$l = (27+18+12+18+12+12+6 / 64) = (105 / 64) \approx 1.83,$$

slightly better then 3:2 compression. In fact an arithmetic code would be applied to a very long input sequence and theabove analysis is applicable to only a single application of the code and not to a sequence of applications as considered with the Huffan code. In partiular, the above code is not uniquely decodable and it does not meet the prefix condition. The code would be improved slightly by realizing that fior a single use, we could shorten several of the words and still be able to successfully decode, e.g., 0100 could be replaced by 010. The example is intended simply to show how an arithmetic code achieves compression, not to provide a practical code.

To achieve compression, one codes long strings of input symbols. This, however, places possible demands on the precision of the aruthmetic as the length of the input sequence grows. Modifying the algorithm to incorporate occasional rescaling and finite precision arithmetis in a consistent way yields an arithmetic code.

We now describe in somewhat more detail the workings of the basic Elias code. The encoder at each time n will look at an input symbol χ_n and then, given its past actions, determine a subinterval

$I_n = [\, a_n, b_n \,)$. It may or may not then output code symbols, depending on what I_n is. Beginning at time $n = 0$, if the first input symbol $x_0 = 0$ set $I_0 = [\, 0,q \,)$. If the first symbol is a 1, set $I_0 = [\, q,1 \,)$. Thus the time 0 subinterval has length equal to the probability of the symbol seen. Note that if we know the first subinterval $I0$ into two furhter subintervals proportional to the input probabilities: $[\, a_0,\ a_0+q(b_0-a_0)\,)$ (having length $(q(b_0-a_0))$ and $[\, a_0+q(b_0-a_0)),\ b_0\,)$. If $\chi_1 = 0$, then I_1 is the first subinterval. Otherwise it is the second symbol χ_1. In addition, it specifies the first subinterval. Otherwise it is the second. Knowing the second subinterval since it is a subset of the first interval. Thus it specifies also the first input symbol. This procedure is then continued, each time dividing the previous subinterval into two subintervals proportional to the input probabilities. The algorithm produces at time n a subinterval of length equal to the probability of the input sequence produced up to that time. Knowing the subinterval In is sufficient to comletely determine the original input sequence up to the nth symbol, i.e., $x_0, \ldots, x_{n-1}$.

The sequence subintervals I_n is itself used to produce the code symbol sequence as follows. For each n, the subintervals endpoints a_n and bn of I_n both have binary expansions. At time 0 check to see if the first term in the binary expansions of a_0 and b_0 agree. This will be the case if either $I_0 \subset [\, 0,\ ½\,)$ or $I_0 \subset [\, ½,\ 1\,)$. If this is the case, the encoder produces the common symbol in the binary expansion, $u_0 = 0$ if $I_0 \subset [\, 0,\ ½\,)$ and $u_0 = 1$ if $I_0 \subset [\, ½,\ 1\,)$. If further binary symbols agree, then these too are released.

If the first symbols in the binary expansions of the interval endpoints do not agree, then no encoder symbol is output and the same test is conducted on l_1. The encoder repeats the test for l_n for increasing n until a_n and b_n of In agree.

In general, at time n, the encoder will have found the largest k for which the first k binary symbols in the binary expansions for a_n and bn

agree and it will have produced the common symbols as the output code symbols $u_0 u_1 \ldots \ldots u_{k-1}$. This condition is equivalent to

$$I_n \subset [\ \Sigma\ u_i\ 2^{-i},\ \Sigma\ u_i 2^{-i} + 2^{-k}\ )$$

formula 10

with k being the largest integer for which the formula holds.

The decoder upon encountering k symbolls from the encoder will know that the above inclusion is true and will be able to reconstruct the corresponding n input symbols. For example, denote the interval on the right hand side by Jk. The decoder tests to see if $Jk \subset [\ 0,q\ )$ or $Jk \subset [\ q,1\ )$. If the former is true, then $x0 = 0$. If the latter is true, $x0 = 1$. One of them must be true since the encoder released symbols. The decoder continues in this way, checking to see if Jk belongs to one of the possible Ij for increasing j (the possible Ij are found by the same recursion used at the input) until it is found that Jk is not a subset of one of the possible Ij, at which point the decoder must wait for more encoded symbols.

The code is noiseless and must look at a variable number of input symbols for each code symbol produced. Although we will not prove it, it can be shown that for an iid input the average number of code symbols produced for each input symbols will converge to the entropy of the source as the encoded sequence becomes long. Unfortunately, however, the code as described is impactible because the precision required to specify the inter val endpoints grows without bound. This defect can be surmounted by modifying the encodeing algorithm to use fixed precision arithmetic. Roughly speaking, one simply computes the needed intervals approximately to within accuracy of the fixed precision arithmetic. In order to avoid overlapping intervals due to the

approximation of the endpoints, a rule is needed to adjust the endpoints so as to produce disjoint intervals. This can be accomplish by suitable scaling and rounding. Although the resulting code no longer yields an average word length exactly equal to the entropy, it can be made arbitrarily close by using sufficiently high precision aruthmetic. Furthermore, the approach can be extanded to sources with memory by carving up the unit interval according to condition of probabilities instead of marginal probabilities. The conditional probabilities can, intern, be estimated from the source itself while coding is going on.

Arithmetic coding is more complicated to implement then Huffman coding, but its compression is typically greater and hence it is a popular approach for entropy coding where the extra compression justifies the extra complexity.

Universal and Adaptive Entropy Coding

Both Huffman codes and arithmetic codes assume a *priori* knowledge of the input probabilities. This information is often not known in practice. Furthermore, the probabilities with time because of nonstationarities of real data, e.g., different computer files may have differing probabilities of 0 and 1, varying from equally distributed (for programs) to highly skewed (for facsmile data). Hence better performance will usually be achieved if a code is flexible or robust in the sense of being able to change according to the local statistical behaviour of the input of being compressed. In other words, a smart code should *adapt* to the source at hand.

Perhaps the earliest approach to adaptive entropy coding was that developed by Robert Rice of JPL and subsequently called the "Rice machine". In his example the input process tended to have two distinct modes with correponding distributions. The modes will remain in effect for long periods of time relative to the codeword sizes. Hence his simple but elegant solution was to design two entropy codes, one for

each mode. Along lock of input symbols could be encoded by simultaneously encoding the input with both codes and seeing which code yielded the most compression. The encoder then send one bit describing which code was used followed by the long encoded sequence to the decoder. The lead bit told the decoder which decoder to use to produce the original sequence. The single bit overhead describing which code to use could be made small (in its constribution to the overall bit per sample) by making the "superblock" length large. This approach to noiseless coding was an early example of what later came to be known as *universal codes*: have a collection of codes matched to different input modes and choose the code which yields the best compression alternatively one can observe the input sequence and guess (or estimates or identify) which mode is in effect, possibly by looking at the histograms or relative frequencies of symbol occurances, and then choose the code designed for that mode. This latter encoder tends to be simpler than the universal encoder if there are many modes, but it might not choose the best code for the input sequence (that is, the code designed for the mode guessed to be in effect might not yield the best compression on the current input sequence). This latter approach, estimating the input statistics and using a code matched to those statistics, is referred to as *adaptive entropy coding*. Clearly the universal and adaptive techniques are intimately related.

Lynch-division and enumerative coding

One of the earliest adaptive codes was a simple and natural means of encoding binary vectors observed by Lynch and Davisson and generalized by cover.the idea is this: Given an input vector dimension *N*,first count the number of 1's and call this number *w(*the *weigth* of the vector). The entrophy encoded vector then consists of a prefix giving

this count (possibly in binary notation for a binary code) followed by an index specifying which of all of the

$$\binom{N}{w} = \frac{N!}{w!\,(N-w)!}$$

possible weight w vectors is input. The references provide simple algorithm for computing this index or *enumaerating* the collection of all weight w vectors. The decoder then reverses the procedure. Note that the determination of w can be viewed as an estimate of an underlying bunary symbol probability since w / N is the relative frequency of 1's in the input sequence. It can be shown that as the vestor size becomes large, this approach performs near the entropy bound if the source is memoryless. In other words, the code is nearly optimal even though its underlying statistics are not known in advance.

Adaptive Huffman coding

In principal it is straightforward tomake the Huffman codes adaptive by combining the ideas. One approach is to absorve a large block of N symbols and estimate the underlying probability distribution from the relative frequencies of the symbols. The estimate of symbol a occuring would then be simply $n_N(a) / N$, where $n_N(a)$ is the count of a's appearence in the block of length N. this probability vector would serve as a prefix to along codeword. Given this vector both encoder and decoder could designan identical Huffman code which would then be used to actually encode the entire data block. If the block is large enough, the overhead information in the prefix would contribute only a small amount to the average bit rate. Although straightforward, this scheme has the obvious drawback of a large delay. In addition, it implicity assumes that the underlying distributions remain constant over the large block size.

Another approach is more adaptive in spirit. Suppose that we have a initial Huffman code based on a *priori* statistics, but we wish to modify the

estimates of these probabilities as more data arrive and to adapt the code correspondingly.

A strategy similar to the previous construction would be as follows. Suppose that at time N-1 we have probability estimates $P_{N-1}(a_i)$ for all the source symbols $a_0, a_1, \ldots, a_{M-1}$,

$$P_N(a) = (n_{N-1}(a_i) / N\text{-}1),$$

along with the corresponding Huffman code. The Nth input symbol $x_N = a$ is then encoded and decoded using this Huffman code and all of the probabilities are updated using the new relative frequencies: since only the count for the symbol a is changed,

$$P_N(a) = (n_N(a) / N) = ((N\text{-}1)P_{N-1}(a) + 1) / N$$

$$PN(a_i) = (n_N(a_i) / N) = ((N\text{-}1)P_{N-1}(a_i) / N \quad \text{if } a_i \neq a$$

These new and improved probabilities are known to both the encoder and decoder, which can be then design a new Huffman code for use on the next input symbol.

Although theoretically a reasonable way to adapt, this code has complexity bordering on the ridiculous. For each new symbol a brand new code must be designed. Furthermore, the increasing precision demanded of the arithmetic as N grows large is not practicable. A more practical approach is found by using the structure of Huffman codes emphasising the counts rather than the relative frequencies. The main idea is simple. Because of the sibling property of Huffman codes, we need not redesign the entire tree at each step. If the probabilty estimates are modified a new symbol so that a change in the ordering results and the sibling property is violated. We only have to do some minor surgery on the tree to regain the sibling property and hence have a Huffman code for the modified probability estimates.

Suppose now that instead of saving the probabilities, we have something proportional to probabilities. For the moment suppose that at each step we associate a set of *weights* w_i aith the source symbols a_i. The weights are nonnegative and we could form explicit estimates of the probabilities by normalizing; that is,

$$P(a_i) = w_i / \sum_{k=0}^{M-1} w_k ,$$

but in fact we will not do this. The weights canbe used in place of probabilities to design a Huffman code and to find the corresponding ordered tree W of siblings guaranteed by the sibling property.

Suppose now that at time N we have (as befor) a Haffman code together with its ordered list, W (now containing the weights instead of probabilities but the order of listing nodes is the same). As before we see a new symbol and we begin to encode it with the currently, available tree. We first select the leaf corresponding to the symbol in the current tree. Befor continuing through the tree, however, we increment the weight of this node by one; that is, the new weight of the node is 1 plus the old weight. We then check the ordered list to see if the weight of this node is still no greater than the node above it in the ordered list. If this is the case, then this part of the tree has not changed and we advance to the parent node of the leaf, producing the branch label on the way. If, on the other hand, the weight of the current node now exceeds that of the node above it on the list, we no longer have an ordered list and the old tree is no longer a Huffman code for the new set of weights. Thus we must rearrange things before advancing. Look above the current node and find the (highest weight) node in the list that has a weight that is less than the current node. Exchange the current node with that higher node, forcing its weight to be listed correctly in the ordered list. This exchange carries with it all of the

corresponding subtree information; that is the current node losses its all parents and inhehrets the parent of the former occupant of the higher place in the list. Likewise the dethroned higher node exchanges parents. (tracking the detailsof the tree exchanges requires an approprimate data structure for storing the trees and is treated in some detail in Gallager). Now that weight here current node has moved up to its appropriate (higher weight) position in the ordered list *W*, we once again have a Haffman code for the new weights (at least up ot the current node in its new position). Thus we can advance to the (new) parents nodes and produce an output symbol, the label of the branch.

We have now either arrived at a new node in the original tree, or exchanged nodes and arrived at a new node in a modified tree. Either way we have produced a symbol and arrived at a new node. The process now repeats. Augment the weight of the new node and check its position in *W*. Either the tree is not changed and we can advance, or we must exchange nodes and advance in a modified tree. This process continues until we arrive at the root, at which time a complete word is produced for the input symbol. note that the weights of all the nodes visited have been augmented by one.

An obvious difficulty with this scheme is that as the number of inputs observed increases, the weights grow without bound. This can be resolved by periodically dividing the weights by some constant to reset them. The period of this rescaling and the size of the constant effectively determine how slowly or rapidly the adaptation algorithm forgets the past.

The program codes "compact" on Unix systems is an adaptive Huffman code based on Gallager's algorithm. As described in the on-line manual, the algorithm compresses text by %38, PASCAL source code by %43, C source code by %36, and binary by %19. Thus the compression is typically less than 2:1.

A final adaptive code which makes no assumptions on the input statistics and which does not explicitly estimate those statistics is of

sufficient importance to merit its own section. The next section is devoted to a simple and elegant technique for noiseless coding due to Ziv and Lempel.

Ziv-Lempel Coding

The final noiseless code that we consider is inherently universal in its operation. It is called the Ziv-Lempel code after its inventors. The code shares the property of the arithmetic codes that variable numbers of input symbols are required to produce each code symbol. unlike both Huffman and arithmetic codes, however, the code does not use any knowledge of the probability distribution of the inputs. As with the Elias code, the Ziv-Lempel code achieves the entropy lower bound then applied to a suitably well-behaved source. Also as with the Elias code, the Ziv-Lempel code is not practicable in its simlest form and varitions are required for real applications that require some loss of optimality.

Once again we sketch the basic idea of the code by means of a binary example. We use a variation of the Ziv-Lempel algorithm suggested by Welch who corrected an error in the original algorithm and whose paper was largely responsible for popularizing the algorithm in the computer science community.

The basic idea is that an input sequence is recursively parsed into nonoverlapping blocks of variable size while constructing a dictionary of blocks seen thus far. The dictionary is initialized with the available single symbols, near 0 and 1. Each successive block in the parsing is chosen to be the argest (longest) word w that has appeared in the dictionary and hence the word wa formed by concatenating w and thefollowing symbol is not in the dictionary. Before continuing the parsing wa is added to the dictionary and a becomes the first symbol in the next block. Before detailing the parsing and dictionary construction, we show how the parsing along works.

Suppose that we see the sequence

$$0110011001011000010011 0$$

parsing this sequence using the above rule yields the following segmentation, where the parsed blocks are enclosed in parantheses and the new dictionary word (the parsed block followed by the first symbol of the next block) written as a subscript:

$$(0)_{01}(1)_{11}(1)_{10}(0)_{00}(10)_{100}(01)_{010}(011)_{0110}(00)_{000}(00)_{001}(100)_{1001}(11)_{110}(0).$$

The parsed data implies a code by indexing the dictionary words in the order in which they were added and sending the indices to the decoder. We now consider these operations in more detail.

For an input sequence $x_0x_1\ldots\ldots$ we will produce an output binary sequence $u_0u_1\ldots\ldots$ The encoder will map a variable number of input symbols into a fixed number, say N, of coded symbols. The N coded symbols should be thought of as an integer between 0 and 2^N-1. Both encoder and decoder will construct a code table of $2N$ entries as the data is processed. The table will consist of variable length input strings, each assigned an integer (or binary N-tuple) codeword. At time 0 the encoder and decoder both have identical tables assigning integers to single input symbols. At step n in the encoding process the encoder will look at the $k(\ n\ )$th input symbol, where the pointer $k(\ n)$ will be made explicit shortly. The encoder then finds the largest integer l for which the sequence $x_{k(n)},\ x_{k(n)+1},\ \ldots,\ x_{k(n)+l-1}$ is contained in its table and hence the seuence $x_{k(n)},\ x_{k(n)+1},\ \ldots,\ x_{k(n)+l}$ is not contaned in the table. The encoder then produces as a codeword the integer index of the existing word $x_{k(n)},\ x_{k(n)+1},\ \ldots,\ x_{k(n)+l-1}$ and also adds the new word $x_{k(n)},\ x_{k(n)+1},\ \ldots,\ x_{k(n)+l}$ to the table, assigning it the next available index.

The encoder then sets the pointer for the next step at $k(n+1) = k(n)+l$, that is, it will begin the next step on the last input symbol considered in the current step.

As an example of encodeing, suppose we begin with *table7* and we wish to encode the binary 011001100101100001001l0. The steps are depicted in *tables 8* through *table 15*.

Input string	index
0	0
1	1

table 8

input string	index
0	0
1	1
01	2

table 9

input string	index
0	0
1	1
01	2
11	3

table 10

input sting	index
0	0
1	1
01	2
11	3
10	4

table 11

input string	index
0	0
1	1
01	2
11	3
10	4
00	5

table 12

input sting	index
0	0
1	1
01	2
11	3
10	4
00	5
011	6

table 13

input string	index
0	0
1	1
01	2
11	3
10	4
00	5
011	6
100	7

table 14

input string	index
0	0
1	1
01	2
11	3
01	4
00	5
011	6
100	7
010	8

table 15

Looking through the sequence of tables, it can be seen that things start slowly as the table builds. At completion of all steps in the tables, the encoder has produced the sequence of integers (or N-dimensional binary vectors)

0110242

Before describing the operation of the decoder we observe an important property of the encoder:

The last-first property: The lastsymbol of the most recent word added to the table is the first symbol of the next parsed vector.

Let us now describe the operation with a table:

in	out	reconstructed sequence	add to table
0	0	$(0)_{0,?}$	
1	1	$(0)_{0,1}(1)_{1,?}$	(0,1) (as 2)
1	1	$(0)_{0,1}(1)_{1,1}(1)_{1,?}$	(1,1) (as 3)
0	0	$(0)_{0,1}(1)_{1,1}(1)_{1,0}(0)_{0,?}$	(1,0) (as 4)
2	(0,1)	$(0)_{0,1}(1)_{1,1}(1)_{1,0}(0)_{0,0}(01)_{0,1,?}$	(0,0) (as 5)
4	(1,0)	$(0)_{0,1}(1)_{1,1}(1)_{1,0}(0)_{0,0}(01)_{0,1,1}(10)_{1,0,?}$	(0,1,1) (as 6)

table 16

Unfortunately, there is a problem with the agorithm as described. There exists a type of sequence which at first glance appears confusing to the decoder. Consider, for example, a ternary source with symbols { 0,*a*,*b* } and an input sequence

$$a000ba0a\ldots\ .$$

The initial code table is

input string	index
0	0
a	1
b	2

table 17

The encoder will parse this sequence as

$$(a)_{a,0}(0)_{0,0}(0,0)_{0,0,b}(b)_{b,a}(a0)_{a,0,a}\ldots.$$

and take the actions depicted as

time	send	new entry	index
1	1 (for 0)	(*a*,0)	3
2	0 (for 0)	(0,0)	4
3	4 (for (0,0)	(0,0,*b*)	5
4	2 (for *b*)	(*b*,0)	6
5	3 (for (*a*,0))	(*a*,0,*a*)	7

table 18

The decoder then begins as previously described and reforms the actions of *table19*. The problem is immediate. The decoder receives an index for the table entry 4, but the entry does not yet exist in the table! Without additional guidence, the decoder is stuck. It turns out that this behaviour can arise whenever one sees a pattern of the form! Without additional guidence, the decoder is stuck. It turns out that this behaviour can arise whenever one sees a pattern of the form *xwxwx*,

where x is a single symbol and w is either empty or sequence of symbols such that xw already appears in the decoder and encoder table, but xwx does not.

The code is noiseless and can be shown to be optimum in the limit of unbounded table size. The disadventage of the algorithm is that unmangeably large tables may be required in some applications in order to achieve the desired performance. Any real application must necessarily have a bound on table size. Once reached, the encoder can no longer add codewords and must simply use the existing dictionary. This dictionary can be used in a statistic fahion to encode the remaining input, or it can dynamically adapt to track varying input behaviour.

CONCLUSION

At the end we can say *entropy coding* is also often referred to as *noiseless coding, lossless coding,* and *data compaction coding.* It is also referred to simply as *data compression* in the computer science literature, but it is avoided that this nomenclature as entropy coding is a very special case of data compression.

Most peaple relative to the computers knows what -zip- and -arj- terms indicate. We use them for compressing data consists of many files into a single file which has less capacity then the original data. And we hide it into either our hard disk or diskettes. Todays with using adventages of Windiws95 a new -zip- technique is generated under the name of -Zif-. I hope you have had an idea about this technique under the title "Ziv-Lempel coding". With the use of this technique the compressed files are stored in a single directory which you may see what it is consisted of by either using 'explorer' programs. And you can also execute a file inside the zif-directory.

We have prepared this project with a sensitive approaches to all the subjects included under title *entropy coding.* With our hard work we have got very useful knowledges.

We hope you would appreciate our hard work and enjoyed reading our report.

REFERENCES

1-COMPUTER CODING OF FUNDAMENTAL:(page 355)

M. L. Gambhir

2- UNIVERSAL & ADAPTIVE ENTROPY CODING :

Robert G. Tracy
Russell S. Fling

3- HUFFMAN CODING : (page 313)

William Brown