COURSE CODE: CIT 413 (2 Units)

COURSE TITLE: DATA COMPRESSION

Unit 1: Information Theory

1.1 Introduction to Information Theory

In 1948, Claude Shannon, a mathematician at Bell Labs, published a paper entitled *A Mathematical Theory of Communication*. The paper caught the immediate attentions of mathematicians and scientists in applied mathematics, electrical engineering, and computer science involving the quantification of information. Several disciplines spun off as the result of reactions to this paper, including information theory, coding theory, and the entropy theory of abstract dynamical systems.

Information theory is a branch of mathematics that overlaps into communications engineering, biology, medical science, sociology, and psychology. The theory is devoted to the discovery and exploration of mathematical laws that govern the behaviour of data as it is transferred, stored, or retrieved. Information theory is based on probability theory and statistics.

Whenever data is transmitted, stored, or retrieved, some variables usually determine the properties of data transmission from sender to recipient. In general, such variables include *bandwidth*, *noise*, *data transfer rate*, *storage capacity*, *number of channels*, *propagation delay, signal-to-noise ratio, accuracy* (or error rate), *intelligibility, and reliability*.

However, in audio systems, additional variables include *fidelity* and *dynamic range*. In video systems, *image resolution*, *contrast*, *colour depth*, *colour realism*, *distortion*, and the number *of frames per second* are significant variables.

One of the most important applications of information theory is to determine the optimum system design for a given practical scenario.

1.2 Description of Information Theory

Since inception of information theory, concern experts continue to proffer different meanings of the theory. Some brief descriptions of information theory are listed below;

i. It is a mathematical theory that quantifies information by showing how to measure information, so that one can answer the question, how much information is included in a given piece of data? With a precise number!

ii. It focuses on how to transmit data most efficiently and economically, and to detect errors in its transmission and reception.

iii. It concerns basic data communication theory that applies to the technical processes of encoding a signal for transmission, and provides a statistical description of the message produced by the code.

1.3 Entropy

The concept of entropy connoting disorder or unpredictability originated in thermodynamics (as the 2nd law) and statistical mechanics, it is applicable in myriad of subjects such as communications, economics, information science, computer science, linguistics, music. In day-to-day life it manifests in many areas of life from household to office when effort is not made to keep things in order (chaos).

Entropy is a key measure of information, which is usually expressed by the *average number of bits needed to store or communicate one symbol in a message*. Entropy quantifies the uncertainty involved in predicting the value of a random variable. It can mean different things in different field as discussed below:

Computing

In computing, entropy is the randomness collected by an operating system or application for use in cryptography or other uses that require random data. This randomness is often collected from hardware sources, either pre-existing ones such as mouse movements or specially provided randomness generators.

Information theory

In information theory, entropy is a measure of the *uncertainty* associated with a random variable. The term by itself in this context usually refers to the **Shannon entropy** discussed in next section.

Entropy in data compression

Entropy in data compression may denote the randomness of the data that you are inputting to the compression algorithm. The greater the entropy, the lesser the compression ratio. That means the more random the text is, the lesser you can compress it.

1.4 Shannon's Entropy

Shannon entropy is a type of entropy which quantifies, in the sense of an expected value, the information contained in a message, usually in units such as *bits*. Equivalently, the Shannon entropy is a measure of the average information content one is missing when one does not know the value of the random variable.

Shannon's entropy represents an absolute limit on the best possible lossless compression of any communication: treating messages to be encoded as a sequence of independent and identically-distributed random variables, Shannon's source coding theorem shows that, in the limit, the average length of the shortest possible representation to encode the messages in a given alphabet is their entropy divided by the logarithm of the number of symbols in the target alphabet.

1.5 Shannon's Measure of Information

Shannon derived the formula below for measuring the amount of information in a message by extending previous research done on same subject by Ralph Hartley in 1928 in Bell Labs.

$$\sum_{i=1}^{r} p_i \log_2 \frac{1}{p_i} = -\sum_{i=1}^{r} p_i \log_2 p_i,$$

where p_i denotes the probability of the ith possible outcome.

Shannon's formula was later used in different cases of entropy, such as conditional, relative, binary etc. The binary entropy function $H_b(p)$ is a special case of entropy. Binary entropy function ($H_b\,(p)$) is an extract of Shannon's entropy defined as:

If U is binary with two possible values u_1 and u_2, $U = \{u_1, u_2\}$, such that $\Pr[U = u_1] = p$ and $\Pr[U = u_2] = 1 - p$, then $H(U) = H_b(p)$

where $H_b\,(p)$ is called the binary entropy function and is defined as:

$H_b(p) = -p \log_2 p - (1 - p) \log_2(1 - p),\ \ p \in [0, 1].$

<u>Exercise</u>

Show that the maximal value of $H_b(p)$ is 1 bit and is taken on for $p = ½$.

Unit 2: Data Compression Concepts

2.1 Introduction to Data Compression

The convergence of the communications, computing and entertainment industries led to explosive growth of multimedia data, including text, audio, images and video which has made data compression a part of everyday life (e.g. MP3, DVD and Digital TV). Large volume of data from the convergence has thrown up a number of exciting new challenges and more opportunities for new applications of compression technologies. In addition to new compression technologies, effective and efficient storing, managing, retrieving and delivering compressed data across IT infrastructure are great concerns.

Data compression is about finding novel ways of representing data so that it takes very little storage, with the proviso that it should be possible to reconstruct (decompress) the original data from the compressed version. The first thing that one learns about compression is that it is not "one size fits all" approach: the essence of compression is to determine characteristics of the data that one is trying to compress (typically one is looking for patterns that one can exploit to get a compact representation). This gives rise to a variety of data modelling and coding (representation) techniques, which is at the heart of compression.

As data compression attempts to reduce file sizes by encoding the files such that fewer bits of storage are used to represent the data contained in them, the opposite is also desirable for data compression to be useful. Therefore, data compression is a reversible process; the original file can be recreated from the compressed representation by a process called *decompression*. The most commonly used data compression programs are *freearc*, *CCM*, *flashzip*, *nanozip*,*WinZip*, *WinRAR*, *zip*, *gzip*, *bzip2*, *7-zip*, *jpeg* and *mpeg* among many others. When the compression is effective, the resulting stream of codes will be smaller than the original symbols.

Data compression is essential when storage space is at a premium or when data needs to be transmitted and bandwidth is at a premium (which is almost always). Although modern storage devices are portable and available in high capacities, up to terabytes (1 x10^{12}), there are still needs to compress data to save space and cost.

Bandwidth and data transfer rate of data transmission channels are limited despite concerted efforts by telecommunication scientists/engineers to improve technologies powering data transfer at cheaper cost. Thus when data compression is used in a data transmission application, the goal is speed. Speed of transmission depends upon the *number of bits sent*, the *time required for the encoder to generate the coded message*, and the *time required for the decoder to recover the original ensemble*

Compression of data is a necessity in most developing countries where cost of data storage is high and data transfer across networks such as Internet is big challenge due poor network infrastructures.

2.2 Components of Data Compression

In general, data compression consists of two activities known as *modelling* and *coding* as shown in the diagram and expression below;

Data Compression = Modelling + Coding

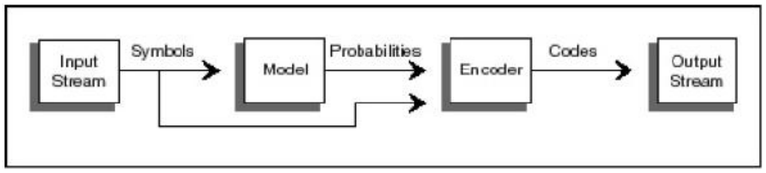


Figure 2.1 Data compression Encoder

Modelling and coding are two distinctly different things. People frequently use the term coding to refer to the entire data-compression process instead of just a single component of that process. You will hear the phrases “Huffman coding” or “Run-Length Encoding,” for example, to describe a data-compression technique, when in fact they are just coding methods used in conjunction with a model to compress data.

Model

Modelling precedes coding and it is mostly statistical techniques that ranks symbols within data file for compression according to their frequency of occurrences or entropies. That is, the decision of the coder to output a certain code for a certain symbol or set of symbols is based on model in use. Many of such statistical techniques have been developed since 1948.

The model is simply a collection of *data* and *rules* used to process input symbols and determine which code(s) to output. Data compression program uses model to accurately define the probabilities for each symbol. The probability generated for each symbol is subsequently used to compute the number of bits required to represent the symbol by coder as expressed below;

Number of bits = $-\log_2(p)$; where *p* is probability of occurrence.

Recall that $\log_2 x = \frac{\log_{10} x}{\log_{10} 2}$

For example a model may generate the number of bits for symbols in table 2.1

Table 2.1

Symbol	**Number of Bits**
e	2
a	2
z	5

Types of Models

A model can be *static* or *adaptive (dynamic)*. In the static case, the compressor analyzes the input, computes probability distribution for its symbols, and transmits the probability distribution to the decompressor followed by the coded symbols. Both the compressor and decompressor select codes of the appropriate lengths using identical algorithms. This method is often used with Huffman coding.

Typically the best compressors use dynamic models and arithmetic coding. The compressor uses previous input to estimate a probability distribution (prediction) for the next symbol without looking at it. Then it passes the prediction and symbol to the arithmetic coder, and finally updates the model with the symbol it just coded. The decompressor makes an identical prediction using the data it has already decoded, decodes the symbol, and then updates its model with the decoded output symbol. The model is unaware of whether it is compressing or decompressing. Modelling techniques will be discussed in detail in subsequent chapter.

Coder

A coder assigns strings of bits to symbols such that the strings can be decoded unmistakably to recover the original data. The coder assigns shorter codes to the more likely symbols. There are numerous efficient and optimal solutions to the coding problem that enable coder to produce an appropriate code based on probabilities received from model. Table 2.2 shows how a typical encoder can represent symbols in a message (column 4) with appropriate number of bits (column 3) to generate compressed data (column 5).

Table 2.2

Symbol	Number of Bits	Bits Pattern	Message	Compressed Data
e	2	01	eaz	01**10***11101*
a	2	10		
z	5	11101		

Since data compression is a reversible process, there is need for decoder to regenerate the original message from compressed data using reverse version of algorithm used by encoder. Huffman coding, arithmetic coding, Asymmetric Binary Coding and Numeric Codes are examples of numerous coding techniques. Some of these coding techniques will explained in other chapters of this course material.

2.3 Classification of Data Compressions

Data compression can be classified into two broad categories; *lossless compression* and *lossy compression* based on the reproducibility of the original data from the compressed data.

Lossy Data Compression

Lossy compression allows for some amount of loss in quality or discarding unimportant data within the acceptable limits as a compromise to save space and resources. The discarded data are those that the human perceptual system cannot see or hear. Thus loss in quality is usually not detectable for all practical purposes. Lossy compression is done mainly on media files such as image, audio and video files.

Lossless Data Compression

Lossless compressions ensure that you can recreate the original file bit for bit in its entirety, which means there is no data loss during the compression and decompression. This type of compression is used mainly for executable programs, source code, text files, data files or certain proprietary formats like x-ray and space exploration.

Lossless compression is used in the popular zip file format and in the Unix tool gzip. Some image file formats, notably PNG, use only lossless compression, while others like TIFF and MNG may use either lossless or lossy methods.

Deciding which data is meaningful is a hard Artificial Intelligence problem that applies to both lossless and lossy compression. Both require a deep understanding of human cognitive psychology.

2.4 Lossy Versus Lossless Compression

i. The advantage of lossy methods over lossless methods is that in some cases a lossy method can produce a much smaller compressed file than any known lossless method, while still meeting the requirements of the application.

ii. Lossy methods are most often used for compressing sound, images or videos with high compression ratio. Audio can be compressed at 10:1 with no noticeable loss of quality, video can be compressed immensely with little visible quality loss, eg 300:1. Loosely compressed still images are often compressed to 1/10th their original size, as with audio, but the quality loss is more noticeable, especially on closer inspection.

iii. When a user acquires a loosely-compressed file, (for example, to reduce download-time) the retrieved file can be quite different from the original at the bit level while being indistinguishable to the human ear or eye for most practical purposes. Many methods focus on the idiosyncrasies of the human anatomy, taking into account, for example, that the human eye can see only certain frequencies of light. Flaws caused by lossy compression that are noticeable to the human eye or ear are known as *compression artifacts*.

iv. In practice, lossy data compression will also come to a point where compressing again does not work, although an extremely lossy algorithm, which for example always removes the last byte of a file, will always compress a file up to the point where it is empty.

v. Lossless compression algorithms usually exploit statistical redundancy in such a way as to represent the sender's data more concisely, but nevertheless perfectly. Lossless compression is possible because most real-world data has statistical redundancy. For example, in English text, the letter 'e' is much more common than the letter 'z', and the probability that the letter 'q' will be followed by the letter 'z' is very small.

vi. Lossless compression schemes are completely reversible so that the original data can be reconstructed, while lossy schemes accept some loss of data in order to achieve higher compression.

2.5 Uses of Data Compression

1. File Transfer

You could benefit from a smaller file size when uploading one or several large files to a website or moving files to another remote machine. This reduces your bandwidth usage as well as the time needed to transfer the files.

2. Backups and Archiving Files

If you have many older files that you do not need to access on an everyday basis, but would like to preserve for other reasons then you want to archive them for safe keeping. These could be old tax records or old photographs or something similar. You can bundle the files together (archive) and compress the archive or compress the files individually without archiving. This also allows you to burn the files onto a CD or DVD for safe keeping while saving on the amount of space required.

3. Web use of media files

If you plan to use the media file on the website, then using a lossy compression on your image and media files will allow you to reduce the size of the files. This can again save on the bandwidth and help load the pages faster.

4. Email

Email servers are notoriously strict on the size of the file attachments that you can have. Compressing the files will reduce the file size and allow you to cram more files into the message as an attachment.

5. File Encryption and Protection

Many simple file formats do not provide much of a file protection or security. This provides a method to secure a complete set the files as they can be encrypted and protected using a single password.

2.6 Data Compression Benchmarks

A data compression benchmark *measures compression ratio over a data set, and sometimes memory usage and speed on a particular computer*. Some benchmarks evaluate size only, in order to avoid hardware dependencies. Compression ratio is often measured by the size of the compressed output file, or in bits per character (bpc) meaning compressed bits per uncompressed byte. In either case, smaller numbers are better. Suppose 8 bpc means original size or no compression, then 6 bpc compressed data means 25% compression or 75% of original size. Generally there is a 3 way trade off between size, speed, and memory usage.

Some existing data compression benchmarks are Calgary Corpus, Large Text Compression Benchmark, Hutter Prize, Maximum Compression, Generic Compression Benchmark, and Compression Ratings.

Exercise

1. Justify the need for data compression especially in Nigeria.
2. Explain the differences between data compression and data archive.
3. List three other data compression benchmarks not mentioned in the course material.

Unit 3: Statistical Modelling

3.1 Statistical Modelling

Statistical Modelling or modelling component of data compression as discussed in Unit 2 uses statistical techniques to generate probabilities for characters in input stream or source file. Once we have a statistical model, coding is a solved problem. But there is no algorithm for determining the best model. This is the hard problem in data compression. Statistical modelling algorithms for text (or text-like binary data such as executables) include: *Burrows-Wheeler transform* (BWT), LZ77 (used by Deflate), LZW and different implementations of Predication by Partial Matching (PPM).

3.2 Context of Model

The context or order of a model is simply the numbers of previous characters from input stream taken into consideration when calculating the probability (predict) of present character. A model of this type is referred to as a *finite context model* or simply *context model*. Thus, if the number of previous characters used to make a prediction is constant, say *i*, then the context is said to be of order *i*.

So when i = 0 (i.e. order-0), then no context is used and the text is coded using unconditioned character counts (i.e., one character at a time). When i = 1, the previous character is used in encoding the current character; when i = 2, the previous two characters are used, and so on. For example in a message “technology”, symbol “h” is said to occur in context “tec” which is order 3, also the symbol “g” is said to occur in context “technolo” which is order 8.

A context model provides a frequency distribution for each context (each character in the order-1 case and each pair of characters in the order-2 case). Context modelling is generally static, semi-static or adaptive (dynamic) so that it compresses the data in a single pass.

3.2.1 Disadvantages of Context Model

i. A disadvantage of context modelling is that the *memory requirement* of the model frequently exceeds the size of the file being compressed - higher order means more the memory requirement. For example, PPMC uses 500 Kbytes of memory to represent a blended order-3 context model of compression ratio of about 30%.

ii. Another disadvantage of context models is that they tend to be much *slower* as the order increases. While some research or production systems may have sufficient processing power to withstand high context order, microcomputer performance may

iii. degrade significantly with same context order.

3.2.2 Types of Context Models

The difference between finite context models is not only the order but how often the probabilities are updated.

Static: A static model uses predefined probabilities, which both the coder and decoder know beforehand (and thus we don't need to transmit them), with those fixed probabilities (we never update them) we code the whole file. This of course has the drawback that if the file doesn't match our probabilities we are going to expand the file instead. For example, if we have probabilities for English text and intend to code Japanese text, the result will be bad. Examples of static probabilities are in tables shown below.

Table 2.1 is a table of order-0 which assigns probability to each English alphabet based on how often it occurs in English text. Table 2.2 is for order 1 where the second number in the first row is the conditional probability *$P(X_i=b|X_{i-1}=a)=0.0228302$*, i.e. the probability of letter "b" occurring given the condition that letter "a" has occurred before "b".

Table 2.3 shows order-3 static probability, The fourth number in the second row is the conditional probability *$P(X_i=d|X_{i-1}=b,X_{i-2}=a)=0.0085877$*, i.e. the probability that letter "d" will occur after letter "b", and lastly after letter "a" (abd). For brevity, we include only the first 3 rows. The entire matrix (729 rows x 27 columns) can be downloaded from http://www.data-compression.com/stat3_out.gz.

Table 2.1: Order-0

A	B	c	d	e	f	... z
0.0651738	0.0124248	0.0217339	0.0349835	0.1041442	0.0197881	...

Table 1.2: Order-1

	A	b	c	d	e	f	...z
a	0.0002835	***0.0228302***	0.0369041	0.0426290	0.0012216	0.0075739	...
b	0.0580027	0.0058699	0.0000791	0.0022625	0.3416714	0.0002057	...
c	0.1229841	0.0000271	0.0215451	0.0005246	0.1715916	0.0000090	...
. . . z							

Table 2.3: Order 2

		A	b	c	d	e	f	...z
a	a	0.0000000	0.0000000	0.0106383	0.3297872	0.0000000	0.0106383	...
a	b	0.0075307	0.0140045	0.0000000	***0.0085877***	0.0202140	0.0000000	...
a	c	0.0066204	0.0000817	0.0963629	0.0006539	0.2368615	0.0000000	...
. . . a	. . . z							

Semi static: This model does two passes, the first is to gather probabilities for all the symbols, then once this is known it outputs the probability distribution and starts to output the codes for every symbol. This model is fast, and in most of the cases it does a good job, however, it has some inefficiencies, among which is that, *it has to pass the whole probability* *distribution (usually this is only applied to order-0, as going to higher orders would need to pass a huge amount of probabilities) or that it does not locally adapt*.

Adaptive: This model processes the symbols in a message one after the other and update model. Thus it has *accurate probabilities* and doesn't need to tell the decoder what the probability distribution is before using it. Of course it has a drawback, *it's slow*, even if there are some schemes for speeding it up (both for Huffman and arithmetic coding). When using this model, we have to update the statistics distributions of the model for each symbol processed.

There are differences in how a model updates its probabilities, for example in most PPM implementations updating only probabilities at *the same and higher orders* where the symbol was found, this is called exclusion, but this is something which changes with every model and will be analysed when they are going to be used. In this course, we shall explain adaptive model using PPM algorithm. PPM programs generally make statistical predictions about upcoming characters based on finite-context models.

Suppose we want to generate adaptive context model for message "ABCABDABE" with order 3 statistics, we also have to generate probabilities for order 2, order 1 and order 0. The probability for the message is shown in table 2.4.

Table 2.4: Order 3 for ABCABDABE

Order 0	**Order 1**	**Order 2**	**Order 3**
Context: "" Symbol Count A 3 B 3 C 1 D 1 E 1	Context: "A" Symbol Count B 3 ------------------------ Context: "B" Symbol Count C 1 D 1 E 1 ------------------------ Context: "C" Symbol Count A 1 ------------------------ Context: "D" Symbol Count A 1 ------------------------ Context: "E" Symbol Count	Context: "AB" Symbol Count C 1 D 1 E 1 ------------------------ Context: "BC" Symbol Count A 1 ------------------------ Context: "CA" Symbol Count B 1 ------------------------ Context: "BD" Symbol Count A 1 ------------------------ Context: "BE" Symbol Count	Context: "ABC" Symbol Count A 1 ------------------------- Context: "BCA" Symbol Count B 1 ------------------------- Context: "CAB" Symbol Count D 1 ------------------------ Context: "ABD" Symbol Count A 1 ------------------------ Context: "BDA" Symbol Count B 1 ------------------------ Context: "DAB" Symbol Count E 1 ----------------------- Context: "ABE"

			Symbol Count

Once we have different orders the question of how to use them arises. One of the solutions is *blending*, which is to combine the probabilities of all the orders to a single probability for every symbol. This is done by adding together the probabilities of the same symbol under different contexts. Using blending in data compression is an alternative of escape codes. In PPM escape code is normally emitted when a symbol is received for the first time from the input stream for a given context, it is stored in a special order denoted as order – (-1).

3.3 Optimal Order for Context Model

Higher orders produce better results, however, it has been proven that orders above 5 are not worth the processing time and space requirements in basic PPM implementations like PPMA or PPMC. However, PPMD, PPM* or PPMZ make use of higher orders to achieve more compression.

PPM as a finite context adaptive model uses arithmetic coding as a coder and tries to accurately predict the next symbol. PPMZ is the most promising model for lossless compression, in the case of infinite context modelling (which is not used) the state of art is Context Tree Weighting (CTW). Unfortunately both of them, especially CTW, are very slow.

There are two major challenges to deal with when using higher orders for context modelling, more context tree (or any other data structure like a suffix tree or hash tables) are required and the number of contexts increase exponentially.

3.3.1 Increase in Data Structures

The simplest finite-context model would be an order-0 model. This means that the probability of each symbol is independent of any previous symbols. In order to implement this model, all that is needed is a single table containing the frequency counts for each symbol that might be encountered in the input stream.

For an order-1 model, you will be keeping track of 256 different tables of frequencies, since you need to keep a separate set of counts for each possible context. Likewise, an order-2 model needs to be able to handle 65,536 different tables of contexts.

3.3.2 Large Context

Also for a higher order there may be too many possible contexts. If our symbols are the 7-bit ASCII codes, the alphabet size is $2^7 = 128$ symbols. Therefore there are $128^2 = 16384$ order-2 context. Similarly, $128^3 = 2097152$ order-3 contexts and so on. The number of contexts grows exponentially, since it is 128^n or, in general, a^n, where a is the alphabet size and n is order of context.

Exercise

1. Draw order 3 context table for MISSISSIPI.
2. Describe how escape code is used in PPM implementation with example.
3. What are the numbers of order-2 and order-3 contexts for an alphabet of size $2^8 = 256$?

Unit 4: Minimum Redundancy Coding and Adaptive Huffman Coding

4.1 Introduction Minimum Redundancy Coding

It isn’t enough to just be able to accurately calculate the probability of characters in a data stream as demonstrated in Unit 3, we also need a coding method that effectively takes advantage of that knowledge to complete data compression process. Probably the best and earliest known coding method based on probability statistics is minimum redundancy coding.

David Huffman was a PhD student at Massachusetts Institute of Technology (MIT), published a paper titled “A Method for the Construction of Minimum Redundancy Coding” in 1952 describing a method of creating a code table for a set of symbols given their probabilities. Minimum redundancy coding improved on major flaw of the suboptimal Shannon-Fano coding (earlier developed by Claude Shannon and Robert Fano) by building the tree from the bottom up instead of from the top down. Minimum redundancy coding is now universally called Huffman coding.

Huffman coding is an *entropy encoding algorithm used for lossless data compression*. The term refers to the use of a variable-length code table for encoding a source symbol (such as a character in a file) where the variable-length code table has been derived in a particular way based on the estimated probability of occurrence for each possible value of the source symbol.

It serves as the basis for several popular programs run on various platforms. Some programs use just the Huffman method, while others use it as one step in a multistep compression process. It works well for text and fax transmissions.

Huffman coding uses a specific method for choosing the representation for each symbol, resulting in a prefix code (sometimes called "prefix-free codes", that is, the bit string representing some particular symbol is never a prefix of the bit string representing any other symbol). The Huffman’s method discussed in section 4.2, *expresses the most common source symbols using shorter strings of bits than are used for less common source symbols*. The bits and the symbols they represent are presented in a table which is guaranteed to produce the lowest possible output bit count possible for the input stream of symbols.

Huffman coding assigns an output code to each symbol, with the output codes being as short as 1 bit, or considerably longer than the input symbols, strictly depending on their probabilities. The optimal number of bits to be used for each symbol is $log_2 \frac{1}{p}$, where p is the probability of a given character. Thus, if the probability of a character is $\frac{1}{256}$, such as would be found in a random byte stream, the optimal number of bits per character is $log_2 \frac{1}{256}$, or 8.

$$log_2 \frac{1}{256} = \frac{log_{10} 256}{log_{10} 2} = \frac{2.408239}{0.301029} = 8.0$$

If the probability goes up to $\frac{1}{2}$, the optimum number of bits needed to code the character would go down to 1.

$$log_2 \frac{1}{2} = \frac{log_{10}2}{log_{10}2} = \frac{0.301029}{0.301029} = 1.0$$

Many variations of Huffman coding exist, some of which use a Huffman-like algorithm, and others which find optimal prefix codes. These variations include, n-ary Huffman coding, Adaptive Huffman coding, Huffman template algorithm, Huffman coding with unequal letter costs, Optimal alphabetic binary trees (Hu-Tucker coding) and canonical Huffman code. This course will discuss adaptive Huffman coding.

Huffman Coding Problems

The first problem with Huffman coding lies in the fact that bit has to be an integral number (that is non-fractional) of bits long. For example, if the probability of a character is $\frac{1}{3}$ the optimum number of bits to code that character is around 1.6. The Huffman coding scheme has to assign either 1 or 2 bits to the code, and either choice leads to a longer compressed message than is theoretically possible.

This non-optimal coding becomes a noticeable problem when the probability of a symbol becomes very high. If a statistical method can be developed that can assign a 90% probability to a given symbol, the optimal code size would be 0.15 bits. The Huffman coding system would probably assign a 1 bit code to the symbol, which is 6 times (i.e. 1/0.15 = 6.66) longer than is necessary.

A second problem with Huffman coding comes about when trying to do adaptive data compression. When doing non-adaptive data compression, the compression program makes a single pass over the data to collect statistics. The data is then encoded using the statistics, which are unchanging throughout the encoding process.

In order for the decoder to decode the compressed data stream, it first needs a copy of the statistics. The encoder will typically prepend a statistics table to the compressed message, so the decoder can read it in before starting. This obviously adds a certain amount of excess baggage to the message.

When using very simple statistical models to compress the data, the encoding table tends to be rather small. For example, a simple frequency count of each character could be stored in as little as 256 bytes with fairly high accuracy. This wouldn’t add significant length to any except the very smallest messages. However, in order to get better compression ratios, the statistical models by necessity have to grow in size. If the statistics for the message become too large, any improvement in compression ratios will be wiped out by added length needed to prepend them to the encoded message.

4.2 Huffman Coding Algorithm

4.2.1 Huffman Encoding

The Huffman encoding algorithm starts by building a list of all the alphabet symbols in descending order of their probabilities. It then constructs a tree, with a symbol at every leaf, from the bottom up. This is done in steps, where at each step the two symbols with smallest probabilities are selected, added to the top of the partial tree, deleted from the list, and replaced with an auxiliary symbol representing the two original symbols. When the list is reduced to just one auxiliary symbol (representing the entire alphabet), the tree is complete. The tree is then traversed to determine the codes of the symbols.

The procedure for building the tree is simple and elegant. The individual symbols are laid out as a string of leaf nodes that are going to be connected by a binary tree. Each node has a weight, which is simply the frequency or probability of the symbol's appearance. The tree is then built with the following steps:

- ➢ The two free nodes with the lowest weights are located.
- ➢ A parent node for these two nodes is created. It is assigned a weight equal to the sum of the two child nodes.
- ➢ The parent node is added to the list of free nodes, and the two child nodes are removed from the list.
- ➢ One of the child nodes is designated as the path taken from the parent node when decoding a 0 bit (left hand). The other is arbitrarily set to the 1 bit (right hand).
- ➢ The previous steps are repeated until only one free node is left. This free node is designated the root of the tree.
- ➢ The tree is then traversed to determine the codes of the symbols.

We shall illustrate Huffman coding with Example 4.1 shown below.

Example 4.1

Given five symbols with probabilities in table 4.1, we will create the binary tree and encoding table as:

Table 4.1: Huffman Coding

Symbol	**Probability**
a_1	0.4
a_2	0.2
a_3	0.2
a_4	0.1
a_5	0.1

1. a_4 is combined with a_5 and both are replaced by the combined symbol a_{45}, whose probability is 0.2.

2. There are now four symbols left, a_1 with probability 0.4, a_2, a_3 and a_{45} with probability 0.2 each. We arbitrarily select a_3 and a_{45}, and combine them and replace them with the auxiliary symbols a_{345}, whose probability is 0.4.

3. Three symbols are now left, a_1, a_2 and a_{345} with probabilities 0.4, 0.2 and 0.4 respectively. We arbitrarily select a_2 and a_{345}. Combine them, and replace them with the auxiliary symbol a_{2345} whose probability is 0.6.

4. Finally, we combine the two remaining symbols, a_1 and a_{2345} and replace them with a_{12345} with probability of 1.

The tree is now completed as shown in Figure 4.1 with height of 4 i.e. *the number if digits in the longest code*. This results in the codes 0, 10, 110, 1110, and 1111.

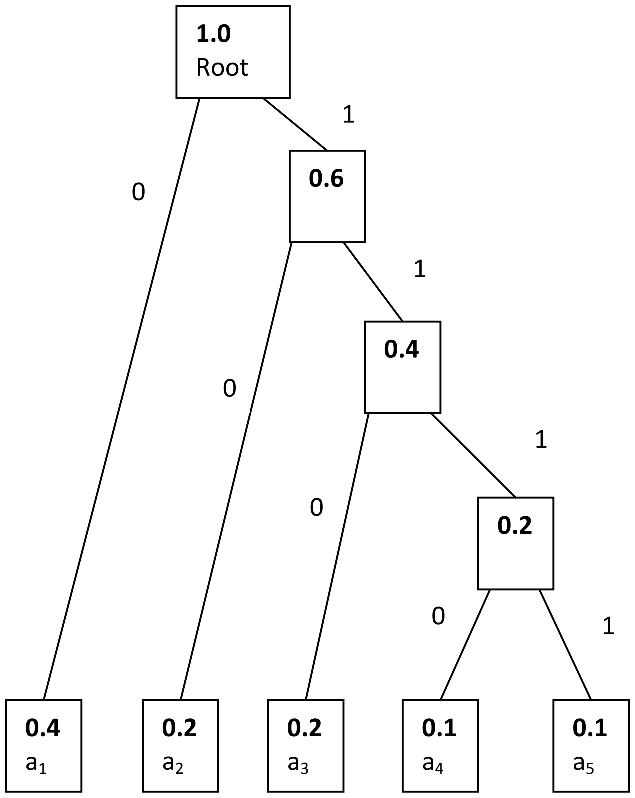


Figure 4.1: Binary Tree

The average size of this code is 0.4 × 1 + 0.2 × 2 + 0.2 × 3 + 0.1 × 4 + 0.1 × 4 = 2.2 bits/symbol, but even more importantly, the Huffman code is not unique. Some of the steps above were chosen arbitrarily, since there were more than two symbols with smallest probabilities.

Table 4.2: Huffman Code

Symbol	Probability	Huffman Code	No of Bits	Size
a1	0.4	0	1	0.4
a2	0.2	10	2	0.4

a3	0.2	110	3	0.6
a4	0.1	1110	4	0.4
a5	0.1	1111	4	0.4
			Average size	2.2 bits/symbol

Choosing Best Arbitrary Pairs

Choosing different arbitrary pairs will either increase or decrease the height of binary tree and of course will give different codes to some symbols, but the bits/symbols will always remain the same. The question is how do we get the best arbitrary pairs? The answer, while not obvious, is simple: The best code is the one with the smallest variance. The variance of a code measures how much the sizes of the individual codes deviate from the average. The variance for Huffman code in table 4.2 is;

$0.4 \times (1 - 2.2)^2 + 0.2 \times (2 - 2.2)^2 + 0.2 \times (3 - 2.2)^2 + 0.1 \times (4 - 2.2)^2 + 0.1 \times (4 - 2.2)^2 = 1.36.$

Suppose we chose another arbitrary pairs different from those in figure 4.1 and got Huffman code in table 4.3 with height of 3, then we can calculate its variance and determine the smaller of the two variances. The table with least variance will generate better Huffman code.

Table 4.3

Symbol	**Probability**	**Huffman Code**	**No of Bits**	**Size**
a1	0.4	11	2	0.8
a2	0.2	01	2	0.4
a3	0.2	00	2	0.4
a4	0.1	101	3	0.3
a5	0.1	100	3	0.3
			Average size	2.2 bits/symbol

The variance for table 4.3 is:

$0.4 \times (2 - 2.2)^2 + 0.2 \times (2 - 2.2)^2 + 0.2 \times (2 - 2.2)^2 + 0.1 \times (3 - 2.2)^2 + 0.1 \times (3 - 2.2)^2 = 0.16.$

Therefore best arbitrary pair chosen yielded the Huffman code in table 4.3 with variance of 0.16 which is smaller than 1.36 generated by code in table 4.2. A simple of thumb to get the least variance is - *when there are more than two smallest-probability nodes, select the ones that are lowest and highest in the tree and combine them.* This will combine symbols of low probability with ones of high probability, thereby reducing the total variance of the code.

4.2.2 Huffman Decoding

Once symbol coding is done and transmitted to decoder, it becomes gibberish at decoder’s end until the process of encoding is reversed. Decoder needs to possess one of the following information generated by coder for decompression to be successful:

i. Frequency or probability distribution of symbols: the probability distribution is transmitted as side information, thereby, adding few hundred bytes to compressed file. Thus decompressor will have to generate Huffman tree and code table. It requires more processing by decompressor as major side effect.
ii. Huffman code table: the Huffman code used to compress the symbol is send along side with code. This requires less processing resources, but increases compressed file size substantially owing to variable length size the each code.
iii. Huffman binary tree: Generated binary tree is written to compressed file. It requires most storage space than two previous approaches mention earlier but least processing resources.

Using any of the information explained above, the decoder will always use or generate Huffman binary tree to decode compressed file with simple algorithm listed below:

- Start at the root and read the first bit off the compressed stream.
- If it is zero, follow the left branch of the tree; if it is one, follow the right branch of the tree.
- Read the next bit and move another branch downward toward the leaves of the tree.
- When the decoder gets to a leaf, it finds the original, uncompressed code of the symbol (normally its ASCII code), and that code is emitted by the decoder.
- The process starts again at the root with the next bit.

We will illustrate this algorithm with example 4.2.

Example 4.2

Suppose we are to use binary tree in figure 4.1 to decode binary string **110101110**. The decoder will start from root of the tree and pick the first digit from the string which is **1**, it will turn right. It will pick the next digit, which is **1** and turn right again. Thereafter, it will pick the next digit which is **0**, this move will bring the decoder to symbol a_3 which is a leaf. The steps followed are represented with dashed line as shown in figure 4.2. Hence the decoder will emit ASCII code for symbol a_3 and move back to the root of the tree. It will pick the next digit which is **1** and continues the steps until another leaf is reached. So the decoded symbols are $a_3a_2a_4$.

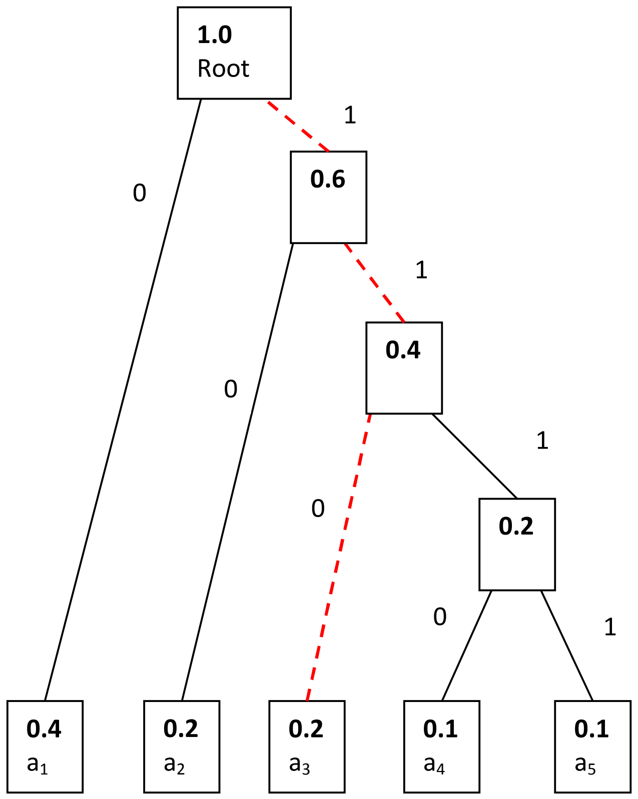


Figure 4.2: Decoding Steps

4.3 Adaptive Huffman Coding

In order to bypass the problems associated with from Huffman Coding in section 4.1, adaptive data compression is called for, thereby leading to Adaptive Huffman Coding. In adaptive data compression, both the encoder and decoder start with their statistical model in the same state. Each of them processes a single character at a time, and updates their models after the character is read in.

Thus adaptive Huffman coding involves calculating the probabilities dynamically based on recent actual frequencies in the sequence of source symbols, and changing the coding tree structure to match the updated probability estimates. It is therefore possible to choose order context for adaptive Huffman coding.

The problem with combining adaptive modelling with Huffman coding *is that rebuilding the Huffman tree is a very expensive process*. For an adaptive scheme to be efficient, it is necessary to adjust the Huffman tree after every character.

Both compressor and decompressor usually start with empty tree and continue to modify binary tree with current statistical model as more symbols are introduced. So, different codes may be generated for same symbol at different time during encoding process. Decompressor will also have to build its tree starting with empty tree to *mirror* or *replicate* the tree of encoder.

4.3.1 Adaptive Huffman Encoding

The standard adaptive Huffman algorithm is called Algorithm FGK named after Faller (1973) and Gallager (1978) who designed the first versions and Knuth (1985) improved the algorithm.

Adaptive Huffman encoding starts with empty tree and build it as symbols are read from input stream. It emits escape code for first occurrence any symbol in a tree followed by uncompressed code (e.g. ASCII) for the symbol and the new symbol is add to new leaf node. The code for escape symbol will continue to change as more new symbols are added. Weight that corresponds to frequency of each symbol is indicated on the leaf node. Maximum of two symbols are combined to form an internal node whose weight is addition of the leaf nodes or other internal nodes below it. All internal nodes or leaf nodes at same level are referred to as *sibling*. *Siblings are arranged in ascending order from left to right, also internal nodes and leaf nodes are arranged in ascending order from bottom up*, this arrangement is called *sibling properties*.

This means that nodes must be listed in the order of decreasing frequency with each node adjacent to its sibling. Thus if A is the parent node of B and C is a child of B, then freq(A)≥freq(B)≥freq(C). The tree may need to be updated (re-arranged) to conform to sibling properties when new symbol is added or frequency of existing symbols in the tree is increased.

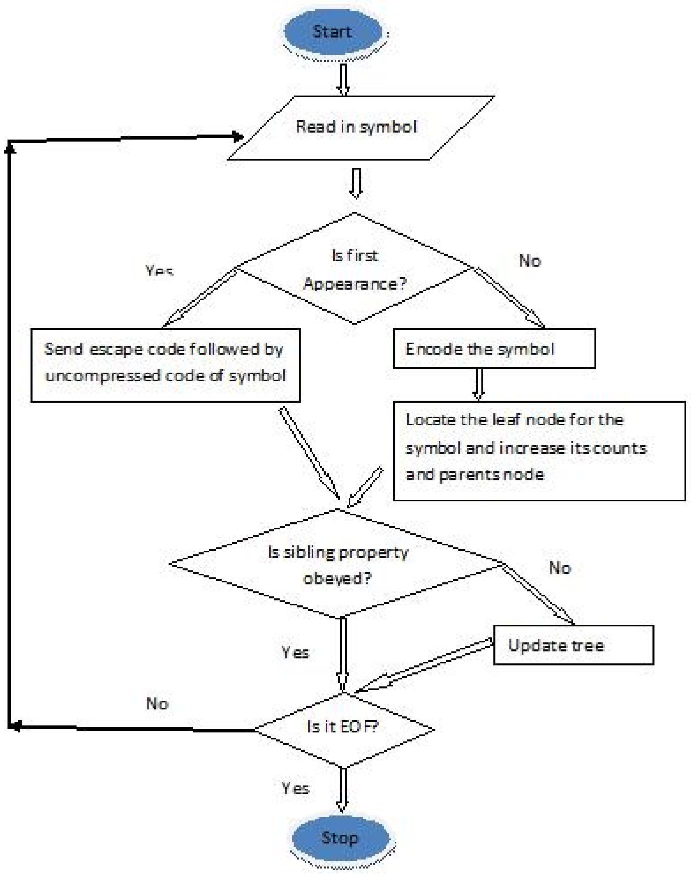

Start
Read in symbol
Is first Appearance?
Yes
No
Send escape code followed by uncompressed code of symbol
Encode the symbol
Locate the leaf node for the symbol and increase its counts and parents node
Is sibling property obeyed?
No
Update tree
Yes
No
Is it EOF?
Yes
Stop

Figure 4.3: Adaptive Huffman Encoding

Example 1

Suppose we want to encode the string "state" and store the encoded string in variable *string_after_coding*. For the first step we need a tree with a special node called escape node. This will look like this:

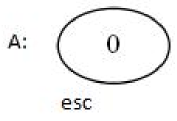


Figure 4.4: Empty Huffman tree

Now let's start to encode the initial string. For this we will need to traverse the string character by character and check if that character is present in our tree. So The first character from string is 's' which isn't present in our tree, will emit uncompressed symbol for it being the first symbol. Therefore the content of *string_after_coding*=**'s'**.

Note that we are operating on a single tree structure and generating the bits string is done before joining the new node with the previous tree. Now, obtaining that code is very easy, we just need to traverse the tree until we get to the escape symbol and after each step we take to the left we add a 0 to the code or 1 if we go right. Symbol 's' will be inserted into new leaf node named s with weight of 1. The tree will now look like:

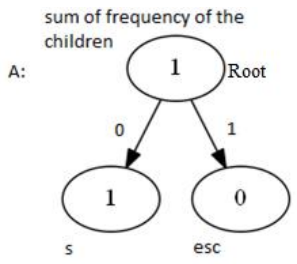


Figure 4.5: Adaptive Huffman with s symbol

Next step is to verify the sibling property. In the tree above, this property is not satisfied so we will change the place of some nodes like this:

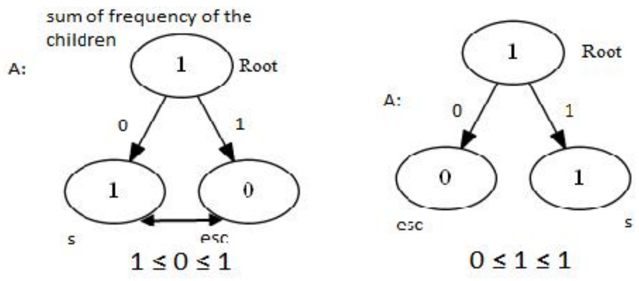


Figure 4.6: Updating Adaptive Huffman Tree

Now we have our tree from the first step and now it’s time to get to the second symbol. As we can see ‘t’ isn’t present in our previous tree so we will do same thing as in first step. This time we will search the tree for current escape node by traversing the tree until we find the escape symbol, which is 0 and emit it before emitting uncompressed code for symbol ‘t’. Therefore *string_after_coding*= **‘s’0’t’**. We will create a new node that will contain the frequency of this new symbol and join this node with our tree resulting to a new tree.The final tree for second step will look like this:

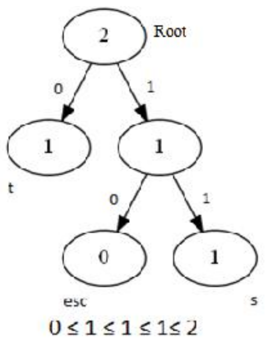
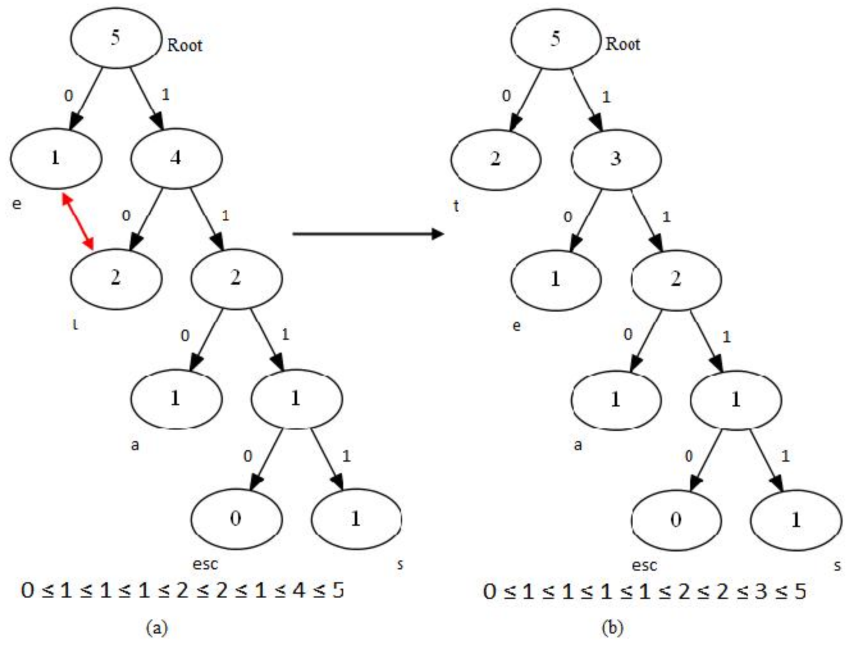


Figure 4.6: Adaptive Huffman with s and t symbols

As you can see the sibling property is satisfied so we don't need to make any changes to the tree.

So we continue to encode the characters one after the other and build the tree (obeying sibling property) until read the last symbol 'e'. The final content of *string_after_coding*= **'s'0′t'10′a'10110′e'** and final Adaptive Huffman tree is shown in (b) below;



Root
e
t
esc
s
a
$0 \le 1 \le 1 \le 1 \le 2 \le 2 \le 1 \le 4 \le 5$
(a)
$0 \le 1 \le 1 \le 1 \le 1 \le 2 \le 2 \le 3 \le 5$
(b)

Figure 4.7: Adaptive Huffman with s ,t ,a and e symbols

Encoding Uncompressed Symbols

Now let's see how to encode each uncompressed symbol from *string_after_coding*= **'s'0′t'10′a'10110′e'**. These symbols are 's',' t', 'a' and 'e'. For this we will need to follow a simple rule of encoding. We consider a set **$L = \{l_1, l_2, l_3, \ldots, l_m\}$** of **m elements** in which l_k is a letter of our alphabet L. For example if we encode ASCII strings each element of the set will be an ASCII code representing the character we are about to encode.

Then we need to pick two numbers, e and r, such that: **$m = 2^e + r$**, where **m is the size of the alphabet and $0 \le r < 2^e$**.

For example, we consider L **= {a, b, c,…, z}** to be the set of English lowercase letters, so **m** is 26 because we have 26 characters. Now we pick e and r to satisfy the formula above: e = 4 and r = 10 because $2^e = 16$ and r is lower than 16. If we took e = 3 then $2^e = 8$ and r will be 18 which is greater than 8. To encode the character we use either rule (i) or (ii) stated below

i. **if $1 \le k \le 2r$, l_k is encoded using (e + 1) number of bits, by calculating binary representation of its position (k – 1) in L;**

ii. Otherwise l_k **is encoded using e number of bits, by calculating binary representation of its position (k – r – 1) in L.**

To encode character ‘a’, k = 1, because ‘a’ is in position 1. Therefore, rule (i) applies; we encode it with 5 bits and get binary representation of k-1 which is 00000. The complete list of characters and their code is presented below:

Character	After encoding	Character	After encoding
a	00000	n	01101
b	00001	o	01110
c	00010	p	01111
d	00011	q	10000
e	00100	r	10001
f	00101	s	10010
g	00110	t	10011
h	00111	u	1010
i	01000	v	1011
j	01001	w	1100
k	01010	x	1101
l	01011	y	1110
m	01100	z	1111

Figure 4.8: Uncompressed Symbols Codes

The ASCII representation of ‘state’ is: 0111 0011 0111 0100 0110 0001 0111 0100 0110 0101; **40 bits**.

Now to generate the final string we will use the previous *string_after_coding* = 's'0′t’10’a’10110’e’ and replace each uncompressed symbol (character) with its corresponding bit representation from the table above. The final string will look like: 1001 0010 0111 0000 0010 1100 0100; **28 bits**.

4.3.2 Adaptive Huffman Decoding

The decoding procedure mirrors the steps of the encoder. It starts with empty tree just like encoder. When it reads the uncompressed form of a symbol, it adds it to the tree and assigns it a code. When it reads a compressed (variable-length) code, it scans the current tree to determine what symbol the code belongs to, and it increments the symbol’s frequency and updates the tree (if necessary) as explained in section 4.3.1. So, the decoder needs to know if the item it has just input is an uncompressed symbol, escape code or a variable-length code. To remove any ambiguity, each uncompressed symbol is preceded by a special, variable-size escape code. When the decoder reads this code, it knows that the next sets of bits are for symbol that appears in the compressed file for the first time.

Huffman coding today is often used as a "back-end" to some other compression methods. DEFLATE (PKZIP's algorithm) and multimedia codecs such as JPEG and MP3 have a front-end model and quantization followed by Huffman coding (or variable-length prefix-free codes with a similar structure, although perhaps not necessarily designed by using Huffman's algorithm[clarification needed]).

Exercise 4

1. Draw the Huffman tree and table for symbols shown in table below. Use the table to encode the word "nigerian".

e	a	g	n	r	i
1	1	1	2	1	2

2. Given the eight symbols A, B, C, D, E, F, G, and H with probabilities 1/30, 1/30, 1/30, 2/30, 3/30, 5/30, 5/30, and 12/30, draw three different Huffman trees with heights 5 and 6 for these symbols and calculate the average code size for each tree.
3. Draw Huffman tree for symbols in table 4.3.
4. Given seven symbols with probabilities .02, .03, .04, .04, .12, .26, and .49, construct binary Huffman code-trees for them and calculate the average code size.
5. Draw adaptive Huffman binary tree for "constitutions".

Unit 5: Arithmetic Coding

Arithmetic coding (AC) bypasses the idea of replacing an input symbol with a specific code. It replaces a stream of input symbols with a single floating-point output number. More bits are needed in the output number for longer, complex messages. This concept has been known for some time, but only recently were practical methods found to implement arithmetic coding on computers with fixed-sized registers.

5.1 Encoding

Arithmetic coding, is entropy coder widely used, the only problem is its speed, but compression tends to be better than Huffman can achieve. *The idea behind arithmetic coding is to have a probability line, 0-1, and assign to every symbol a range in this line based on its probability, the higher the probability, the higher the range which is assigns to it*. Once we have defined the ranges and the probability line, start to encode symbols, every symbol defines where the output floating point number lands. Let's say we have:

Table 5.1: Probability Distribution

Symbol	Probability	Symbol Interval
a	2	[0.0 , 0.5)
b	1	[0.5 , 0.75)
c	1	[0.7.5 , 1.0)

Note that the "[" means that the number is also included, so all the numbers from 0 to 5 belong to "a" but 5. And then we start to code the symbols and compute our output number.

The algorithm to compute the output number is:

- *low = 0*
- *high = 1*

Loop. (For all the symbols).

- ***range = high - low***
- ***high = low + range * high_symbol_interval of the symbol being coded***
- ***low = low + range * low_symbol_interval of the symbol being coded***
- ***write the symbol where the low value falls within the symbol interval***

End loop(When EOF)

Where:

- *range, keeps track of where the next range should be.*
- *high and low, specify the output number.*

And now let's see an example:

Table 5.2: Encoding with AC

Symbol	Range	Low value	High value
		0	1
b	1	0.5	0.75
a	0.25	0.5	0.625
c	0.125	0.59375	0.625
a	0.03125	**0.59375**	0.609375

Table 5.3: Probability Distribution

Symbol	Probability	Symbol Interval
a	2	[0.0 , 0.5)
b	1	[0.5 , 0.75)
c	1	[0.75 , 1.0)

The output number from encoder will be **0.59375** and the frequency distribution table.

5.2 Decoding

In order to decode digits received from encoder, the decoder will first check to see where the number lands, output the corresponding symbol, and then extract the range of this symbol from the floating point number. The algorithm for extracting the ranges is:

- ***Loop (For all the symbols).***
    - ***Locate where number falls within symbol interval and write the symbol***
    - ***range = high_value of the symbol - low_value of the symbol***
    - ***number = number - low_value of the symbol***
    - ***number = number / range***

End Loop(When EOF)

Or:

To eliminate the effect of symbol X from the code, the decoder performs the operation ***Code:= (Code-LowRange(X))/Range(X)***, where Range is the width of the sub-range of X. Table 2.50 summarizes the steps for decoding our example string (notice that it has two rows per symbol).

And this is how decoding is performed:

Table 5.4: Decoding With AC

Number	Symbol	Range
0.59375	b	0.25
0.375	a	0.5
0.75	c	0.25
0	a	0.5

Table 5.5: Probability Distribution

Symbol	Probability	Symbol Interval
a	2	[0.0 , 0.5)
b	1	[0.5 , 0.75)
c	1	[0.75 , 1.0)

You may reserve a little range for an EOF symbol, but in the case of an entropy coder you'll not need it (the main compressor will know when to stop), with and stand-alone compressor/decompressor you can pass to the decompressor the length of the file, so it knows when to stop.

5.3 Arithmetic Coding Challenges

The steps or algorithms stated above for arithmetic encoding and decoding are fairly simple. However, some computing resources constitute limitations that hinder implementations of the algorithms. Among these challenges is the fact that fractional (decimal) part use for encoding grows exponentially and most computers support only 80 bits for decimal number representation. Another challenge associated with arithmetic coding is called *underflow*. Underflow occurs when both high and low get close to a number but theirs most significant bit don't match: High = 0.300001, Low = 0.29997, if we ever have such numbers, and the low and high numbers continue getting closer and closer we'll not be able to output the most significant bit.

Exercise

1. Write short note on how to handle challenges attributed to arithmetic coding.
2. Encode the message “suspension” using arithmetic coding.
3. Generate decoding table for number computed in question 2 above.

Unit 6: Dictionary-Based Compression

6.1 Introduction to Dictionary-Based Compression

So far, the compression methods we have looked at used a statistical model (Huffman Coding and Arithmetic coding) to encode single symbols. They achieve compression by encoding symbols into bit strings that use fewer bits than the original symbols. The quality of the compression goes up or down depending on how good the program is at developing a model. The model does not only have to accurately predict the probabilities of symbols, it also has to predict probabilities that deviate from the mean. More deviation achieves better compression.

Dictionary-based compression algorithms use a completely different method to compress data. This family of algorithms does not encode single symbols as variable-length bit strings using statistical models. They select strings of symbols from the input, search for matches between the strings of symbols to be compressed and a set of strings contained in a data structure (called the 'dictionary') maintained by the encoder. When the encoder finds such a match, it substitutes a reference to the string's position in the dictionary.

The dictionary used in dictionary-based compression holds strings of symbols, and it may be *static* or *dynamic* (adaptive). The substituted symbols are then encoded with *index* (static dictionary) or as a *token* (dynamic dictionary). If the tokens are smaller than the phrases they replace, compression occurs. The tokens form an index to a phrase dictionary.

In many respects, dictionary-based compression is easier for people to understand and its simplicity makes it very popular. Also dictionary-based compression is general purpose, performing compressions on images and audio data as well as on text inputs.

6.2 Static and Adaptive Dictionary-Based Compressions

6.2.1 Static Dictionary

Static dictionary contains full set of strings determined before coding begins and does not change during the coding process. Some dictionaries are permanent, sometimes allowing the addition of strings but no deletions. Static dictionary approach is most often used when the message or set of messages to be encoded is fixed and large.

Static dictionary based compression are mostly ad hoc, implementation dependent, and not general purpose. Also dictionary must be available to both encoder and decoder during processing. When a match is found between word in a message and that in static dictionary, the encoder will simply emit the index assigned to that word in the dictionary to output stream. The same dictionary will also be used by decoder to get uncompressed word.

For example a static dictionary can be embedded in programming language compiler to compress source code, such that new syntax can be added to the dictionary for downward compatibility.

6.2.2 Dynamic Dictionary

Dynamic dictionaries are more common methods. In such method the dictionary may be empty at the start of compression or starts in some predetermined state but the contents change during the encoding process, based on the data that has already been encoded. Both the LZ77 and LZ78 algorithms work on this principle.

Dynamic dictionary compression consists of a loop, each iteration starts by reading the input and breaking it up (parsing it) into words or phrases. It then searches the dictionary for each word and, if a match is found, it writes a token on the output. Otherwise, the uncompressed word is output and also added to the dictionary. The last step in each iteration checks whether an old word should be deleted from the dictionary. This may sound complicated, but it has two advantages:

1. It involves string search and match operations, rather than numerical computations. Many programmers prefer that.

2. The decoder is simple (this is an asymmetric compression method). It reads the next input item and determines whether it is a token or raw data. A token is used to obtain data from the dictionary and write it on the output. Raw data is output as is. The decoder does not have to parse the input in a complex way, nor does it have to search the dictionary to find matches. Many programmers like that, too.

Unit 7: Sliding Window Compression (LZ77)

7.1 Introduction to Sliding Window Compression

The History of modern dictionary-based compression can be traced to the 1977 Ziv and Lempel paper, "*A Universal Algorithm for Sequential Data Compression*," published in *IEEE Transactions on Information Theory* gave insight to sliding window data compression (a dictionary-based data compression algorithm). Initially, this compression method (referred to hereafter as ***LZ77***) does not seem particularly remarkable.

Sliding window compression is an algorithm that represents groups of characters that occur frequently more efficiently. For instance the message, "can count countville count the countably infinite uncountable" repeats the string "count" five times. Sliding window will represent the phrase "count" with a small code or *token* irrespective of location of word "count" within a message.

The algorithm builds a dictionary comprising of previously seen words and use a token assigned to words in the dictionary to code subsequent occurrences of the words. Similar dictionary will also be available to decompression algorithm so that tokens generated by compression and decompression algorithms will be identical.

Before implementing a sliding window compression scheme the size (number of possible entries) of the dictionary must be fixed. More dictionary entries mean a greater chance that a repeated string will be found in the dictionary text and therefore compressed. However, more dictionary entries also lead to longer dictionary codes and more memory utilisation. Thus, old entries in the dictionary must be expunged to minimize memory.

Several improvements were made to LZ77 shortly after the inventors announced their dictionary-based compression method. Compression methods such as LZSS (developed by Storer and Szymanski in 1982), QIC-122 (http://www.qic.org/html), LZX (Jonathan Forbes and Tomi Poutanen) and LZ78 are all variants of LZ77.

Today many applications use LZ variant as core compression algorithm. GIF (Graphics Interchange Format) is a graphics file format that uses a variant of LZW to compress the graphics. The popular RAR software is the creation of Eugene Roshal, who started it as his university doctoral dissertation. RAR is an acronym that stands for Roshal ARchive (or Roshal ARchiver). RAR/WinRAR use LZ with a large search buffer (up to 4 Mb) combined with Huffman codes.

7.2 LZ77 Encoding

The encoder maintains a window to the input stream and shifts the input in that window from right to left as strings of symbols are being encoded. Thus, the method is based on a sliding window. The window below is divided into two parts. The part on the left is *the search buffer*. This is the current *dictionary*, and it includes symbols that have recently been input and encoded. The part on the right is the *look-ahead buffer*, containing text yet to be encoded. Search buffer is assigned more memory spaces, some thousands of bytes long, while look-ahead buffer is only tens of bytes long.

The encoder picks a word from look-ahead buffer and tries to get a match for it in the search buffer, if multiple matches are found the leftmost match in search buffer is selected and used to generate a token for encoding the word in the look-ahead buffer. If a match is not found then raw characters of the word is in the token to encode the new word. Encoded word in look-ahead buffer is moved into search buffer and the numbers of characters corresponding to the encoded word is removed from search buffer through the left end to accommodate new word since its size if fixed. Hence this data compression is called sliding window. The contents of search buffer often change as compression continues and its size is exhausted.

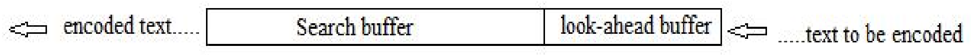


Figure 7.1: LZ77 Buffers

Start of Tokenization

Most implementations of LZ77 start with empty or partially empty search buffer or dictionary and look-head buffer is filled to capacity with words from input stream. The first symbol of the first word will be read from look-ahead buffer into search buffer, but since this is the first symbol in search buffer coder will simply emit raw symbol. Token contains three information the offset, length and next symbol to be encoded, arranged as (**offset**, **length**, **next symbol**), where:

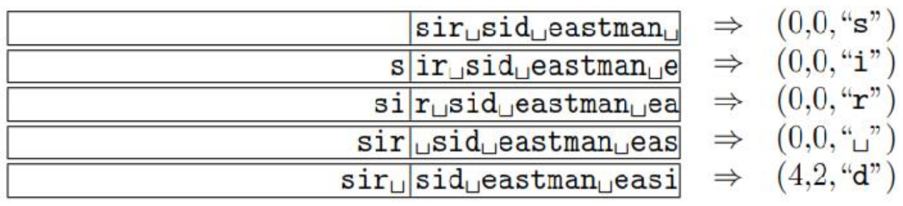
offset - is a number representing start of matched symbol or word position. The numbering is done from right to left of search buffer.

length – in the number of symbols in look-ahead buffer that match symbols in search buffer

next symbol – is the next symbol after the current match.

Hence, tokens with zero offset and length are common at the beginning of any compression job, when the search buffer is empty or almost empty.

Suppose we are to encode message "***sir sid eastman easily teases sea sick seals.....*** ". The first five steps in encoding our example are the following:



	sir␣sid␣eastman␣	⇒	(0,0,"s")
s	ir␣sid␣eastman␣e	⇒	(0,0,"i")
si	r␣sid␣eastman␣ea	⇒	(0,0,"r")
sir	␣sid␣eastman␣eas	⇒	(0,0,"␣")
sir␣	sid␣eastman␣easi	⇒	(4,2,"d")

Figure 7.1: Initial Token

Thus, if backward search yields no match, an LZ77 token with zero offset and length and with the unmatched symbol is written. This is also the reason a token has a third component.

Depending on the size of buffers, a typical token in LZ77 will assign 11, 5, and 8 bits to offset, length and next symbol (ASCII) respectively; therefore, LZ77 token will have a length of 24 bits. Token (0,0,"s") will be 0000000000 00000 01110011, and (4,2,"d") is 0000000100 00010 01100100.

Matching Symbols or Words

The search buffer will definitely get filled up as more words are read in from look-ahead buffer, so some words will have to move out of it. We assume the current status of the buffers in figure 7.1 above is displayed in figure 7.2 after encoding some characters. Suppose we wish to encode more words, the vertical bar between the **t** and the **e** below represents the current dividing line between the two buffers.

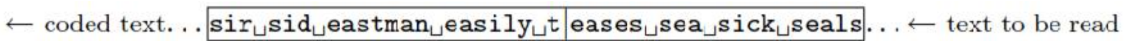


Figure 7.2 LZ77 Encoding

We assume that the text "***sir sid Eastman easily t***" has already been compressed, while the text "eases sea sick seals" still needs to be compressed.

The encoder scans the search buffer backwards (from right to left) looking for a match for the first symbol **e** in the look-ahead buffer. It finds one at the **e** of the word **easily**. This **e** is at a distance (offset) of 8 from the end of the search buffer. The encoder then matches as many symbols following the two **e**'s as possible. Three symbols **eas** match in this case, so the length of the match is 3. The encoder then continues the backward scan, trying to find longer matches. In our case, there is one more match, at the word **eastman**, with offset 16, and it has the same length. The encoder selects the longest match or, if they are all the same length, the last one found, and prepares the *token* **(16, 3, e)**.

This token is written on the output stream, and the window is shifted to the right (or, alternatively, the input stream is moved to the left) four positions: three positions for the matched string and one position for the next symbol, as shown in figure7.3.

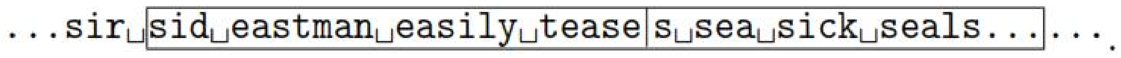


Figure 7.3: LZ77 Tokenising

Selecting Best Match

Selecting the last match, rather than the first one, simplifies the encoder, because it only has to keep track of the last match found. It is interesting to note that selecting the first match, while making the program somewhat more complex, also has an advantage. It selects the smallest offset. It would seem that this is not an advantage, because a token should have room for the largest possible offset. However, it is possible to follow LZ77 with Huffman, or some other statistical coding of the tokens, where small offsets are assigned shorter codes. This method, proposed by Bernd Herd, is called LZH. Having many small offsets implies better compression in LZH.

7.3 Decoding LZ77

The decoder is much simpler than the encoder (LZ77 is therefore an asymmetric compression method). It has to maintain a buffer, equal in size to the encoder's search buffers. The decoder inputs a token, finds the match in its buffer, writes the match and the third token field on the output stream, and shifts the matched string and the third field into the search buffer, as shown in figure 7.4.

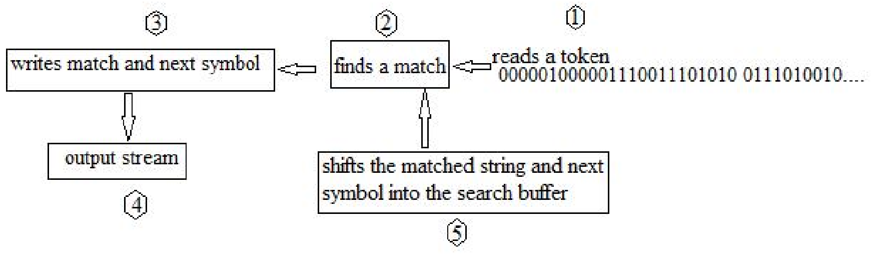

③
writes match and next symbol
②
finds a match
①
reads a token
000001000001110011101010 0111010010....
output stream
④
shifts the matched string and next symbol into the search buffer
⑤

Figure 7.4: LZ77 Decoding

This implies that LZ77, or any of its variants, is useful in cases where a file is compressed once (or just a few times) and is decompressed often. A rarely-used archive of compressed files is a good example.

Problems of LZ77

The first problem to deal with arises when encoding, it has to perform string comparisons against the look-ahead buffer for every position in the search buffer. As it tries to improve compression performance by increasing the size of the search buffer, and thus the dictionary, this performance bottleneck only gets worse.

A second performance problem occurs with the way the sliding window is managed. For conceptual convenience, the discussion here treated the sliding window as though it were truly sliding "across" the text, progressing from the end of the buffer to the front as the encoding process was executed.

References:

Salamon, D. (2008). *A concise introduction to data compression*. London: Springer.

Salamon, D. (2004). *Data compression: The complete reference*. (4th Ed.). London: Springer.

Nelson, M., & Gailly, J. (1995). The *Data compression book*. (2nd Ed.). New York: M&T Books.

Hankerson, D., Harris, G.A., & Johnson, P.D.(2003). *Introduction to information theory and data compression*. (2nd Ed.), London: A CRC Press Company.

http://www.lostsaloon.com/technology/what-is-data-compression-and-why-use-it/

http://en.wikipedia.org/wiki/Huffman_coding

http://marknelson.us/1991/02/01/arithmetic-coding-statistical-modeling-data-compression/

http://chiranjivi.tripod.com/Finite.html

http://static.usenix.org/event/usenix01/full_papers/kroeger/kroeger_html/node4.html

http://www.data-compression.com/theory.shtml

http://www.arturocampos.com/ac_ppmc.html

http://www.arturocampos.com/ac_arithmetic.html

http://www.arturocampos.com/articles.html

http://www.hugi.scene.org/online/coding/hugi%2019%20-%20cofinite.htm