Chapter 8

Information, Entropy, and Coding

8.1 The Need for Data Compression

To motivate the material in this chapter, we first consider various data sources and some estimates for the amount of data associated with each source.

- **Text** Using standard ASCII representation, each character (letter, space, punctuation mark, etc.) in a text document requires 8 bits or 1 byte. Suppose we have a fairly dense text with 50 lines of text per page and 100 characters per line. This would give 5 kB per page. If the text were a hefty 1000 pages, this would result in 5 MB. Alternatively, if we assume one word is on average 6 characters, then each word requires approximately 6 bytes. Therefore, a one million word document would require approximately 6 MB. Although these numbers may sound big, remember that this is for a very large text document. As we'll see other media result in significantly more data.

- **Audio** In digital audio, it's typical to use 16 bits per sample and 44,100 samples per second, resulting in 706 kb/sec or 88.2 kB/sec. With two channels for stereo we would get 176 kB/sec. Therefore, just 30 seconds of digital audio requires almost 5.3 MB, which is more than the 1000 page text document. An hour of two-channel audio results in about 635 MB.

- **Images** For a gray scale image, it's typical to use 8 bits/pixel. A good resolution image might be 512×512 pixels, which would result in 0.26 MB. For a color image, we need to multiply by 3 for the three color components, resulting in about 0.786 MB. Assuming our previous estimate, of about

†Lecture Notes for ELE201 Introduction to Electrical Signals and Systems.
‡Thanks to Keith Vallerio for producing the figures.

6 characters per word, this means such an image is worth more 100,000 words, rather than 1,000 words! Only 7 such images would result in about 5.5 MB, more than the 1000 page text document.

- Video A standard frame rate for video is about 30 frames/sec. Using the estimates for images, we get about 7.86 MB/sec for black/white video and about 23.6 MB/sec for color video. This means that just one second of black/white video or 1/4 of a second of color video results in more data than the 1000 page text document. One minute of black/white video requires about 472 MB, while one minute of color video requires 1.416 GB. A two hour black/white video requires 56.6 GB, while a two hour color video requires almost 170 GB!

The estimates above assume the data is represented directly in raw format (e.g., one byte for each character, one byte per gray scale pixel, etc.). The goal of compression is to represent signals (or more general data) in a more efficient form. Compression is extremely useful for both storage and transmission.

Of course, the need for compression is dependent on storage and communication technology. The estimates above are best appreciated in the context of typical capacities of storage media and data rates for communication. Some typical digital storage media and their capacities are floppy disk (about 1.4 MB), zip disk (100 MB or 250 MB), CD (about 700 MB), DVD (4.25 GB single-sided, 8.5 GB double-sided, 18 GB double-sided and dual density). Some typical data rates for wired communication are 56 kb/sec for phone line modems, ...

Although capacities of storage media and communication data rates available for most users have been increasing rapidly, the demand for data has kept pace. So far, it appears that "if the capacity is available the bits will come". Moreover, the demand for throughput in wireless communication has been rapidly increasing and far outpacing the increases in efficiency in use of the limited spectrum. These forces will ensure that compression remains an important topic for the foreseeable future.

8.2 Basic Idea of Compression

The basic idea of compression is to exploit (in fact, remove) redundancy in data. Compression can be broken down into two broad categories. In *lossless* compression, from the compressed data one is able to reproduce the original data exactly. In *lossy* compression, the original data cannot be reproduced exactly. Rather we allow some degradation in the reproduced data.

In a sense, sampling and quantization can be considered as forms of compression. Sampling is lossless if we sample above the Nyquist rate, and is lossy otherwise. Quantization is lossy compression. Clearly there is a tradeoff between the quality and amount of data. In sampling and quantization, the signal is converted from an analog to a digital signal. This "compression" has the dramatic effect of converting infinitely many samples each of which requires infinite precision to a finite number of samples that each have finite precision.

However, even after a signal is digitized, we can often compress the data still more. This is the usual goal of data compression. Thus, in this and the next chapter, we assume that we already have digital data, and we discuss theory and techniques for further compressing this digital data.

8.3 The Coding Problem

In this section, we formulate the general "coding problem", so that the ideas we discuss will be applicable to general digital data rather than being developed for one specific application.

Suppose we are given M symbols denoted $s_1, s_2, \ldots, s_M$. For example, for images and video we might have $M = 256$ with the symbols s_i denoting the 256 gray levels. Or in a certain application, we might have $M = 26$ with the s_i denoting the 26 letters of the alphabet, or $M = 10$ with the s_i denoting the 10 digits. For general text files, we might have $M = 128$ (as in ASCII) with the s_i denoting 128 characters including upper and lower case letters, digits, punctuation marks, and various special symbols. Regardless of the basic symbols being used for the application of interest (letters, gray levels, etc.), we would like to represent these symbols using bits (i.e., just 0's and 1's). The bit strings used to represent the symbols are called the codewords for the symbols. The coding problem is to assign codewords for each of the symbols $s_1, \ldots, s_M$ using as few bits per symbol as possible.

How many bits do we need per symbol? The obvious answer is that we need $\log_2 M$ bits. For example, if there are 8 symbols $s_1, \ldots, s_8$, then we can use the codewords 000, 001, 010, 011, 100, 101, 110, 111. Likewise, for 256 gray levels we need $\log_2 256 = 8$ bits/symbol. For 26 letters, $\log_2 26 \approx 4.7$, which means we need 5 bits/symbol. With 4 bits we can only represent 16 unique symbols. Is $\log_2 M$ the best we can do?

First, note that implicit in our discussion so far is that each codeword must correspond to a unique symbol. Otherwise, the representation would not be very useful since we would not be able to recover the original symbols (and we're interested in lossless coding). We will say more about this and other conditions shortly.

Subject to the unique representation alluded to above, we cannot in general do any better than $\log_2 M$ without additional assumptions. However, if there's some structure in the data source or if we know something about the symbols we expect to encounter, then we *can* do better. The simplest example of such knowledge/structure is if some symbols are more likely than others, but are otherwise assumed to be random with each symbol being generated independently of other symbols. We will discuss this case in detail as it elucidates the main ideas. However, researchers in the field of information theory and coding have developed results in much more general settings.

8.4 Variable-Length Coding

Assuming that some of the symbols are more likely than others (and assuming we know the respective probabilities of occurrence), the key idea to obtaining a more efficient coding is to use *variable-length coding*. That is, we assign shorter codewords to the more frequently occurring symbols and longer codewords to the symbols that occur infrequently. The standard coding that uses the same number of bits for each codeword is called fixed-length coding.

Variable-length coding is a natural idea and has been used for some time. For example, Morse code exploits this idea in the relative frequency of occurrence of letters in the English language (see Figure XX). As another simple example, consider the following.

Consider four symbols s_1, s_2, s_3, s_4. With the standard (fixed-length) encoding we would need 2 bits/symbol – e.g., we might assign the codewords 00, 01, 10, 11, respectively.

Suppose we know that the probabilities for the symbols are 1/2 for s_1, 1/4 for s_2, and 1/8 each for s_3 and s_4. How might we exploit this knowledge?

Suppose we assign the codewords 0, 10, 110, and 111 for s_1, s_2, s_3, and s_4, respectively. In this case, occasionally we do better than the standard encoding (using only 1 bit for s_1 instead of 2 bits). However, sometimes we also do worse (using 3 bits for s_3 and s_4 instead of 2 bits). To precisely compare the new code with the standard encoding, we can compute the average number of bits/symbol of the codes. The standard coding *always* uses 2 bits, so obviously the average number of bits per symbol is also 2. For the new code, how should we compute the average? We should use the probabilities of the symbols, of course! We find that the average length (bits/symbol) is

$$(1)(1/2) + (2)(1/4) + (3)(1/8) + (3)(1/8) = 1.75.$$

Thus, on average we only need 1.75 bits/symbol instead of 2 bits/symbol. For a file with 1 million symbols, this would mean we only need 1.75 Mbits with the variable-length code instead of 2 Mbits with the standard (fixed-length) code.

8.5 Issues in Variable-Length Coding

With variable-length codes, the issue of codewords corresponding to "unique" symbols is a little more subtle than with fixed-length codes. Even if there is a unique correspondence, another subtlety can arise in decoding. We now discuss these issues.

Unique Decodability With variable length codes, in addition to not having two symbols with the same codeword, we also have to worry about some combination of symbols giving the same string of bits as some other combination of symbols. For example, several symbols with short codewords could combine to form a bit string corresponding to a longer codeword for some other symbol.

Suppose for four symbols s_1, s_2, s_3, s_4 we assign the codewords 0, 10, 01, 11, respectively. Then we can't tell whether 0110 corresponds to $s_3 s_2$ or $s_1 s_4 s_1$. This shows that this particular code is not uniquely decodable.

A code is said to be *uniquely decodable* if two distinct strings of symbols never give rise to the same encoded bit string. Equivalently, any bit string can arise from at most one string of symbols. Note that it is possible that *no* string of symbols could give rise to a given bit string. For example, if the codewords for s_1, s_2, s_3, s_4 are 000, 001, 010, 011, respectively, then the string 111 can never arise from any string of symbols.

Unique decodability is an absolute must if we wish to have useful (and lossless) compression. The following property, though not a "must", is useful, and, as it turns out, can be achieved at no cost.

Instantaneous/Prefix-Free Codes A code is called *instantaneous* if each symbol can be decoded as soon as the corresponding codeword is completed. That is, it is not necessary to see bits of later symbols in order to decode the current symbol.

For example, the variable-length code of Example XX is instantaneous. But the code 0, 01, 011, 111 for the symbols s_1, s_2, s_3, s_4, respectively is *not* instantaneous (although it *is* uniquely decodable). To see this, consider the bit stream $011111\cdots1$. We can't tell if the first symbol is s_1, s_2, or s_3, although it's clear that after this first symbol we have a sequence of s_4's. Once we see the last 1, we can then work backwards to eventually find out what was the first symbol.

It turns out that being an instantaneous code is equivalent to a property known as being prefix-free. A code is said to be *prefix-free* if no codeword is the prefix (the first part) of another codeword. It's easy to see that the code of Example XX is prefix-free, and so the code is also instantaneous. However, for the code above (0, 01, 011, 111), the codeword for s_1 is a prefix of the codewords for s_2 and s_3. Likewise the codeword for s_2 is a prefix of the codeword for s_3. Therefore, this code is *not* prefix-free, and as we saw above is also not instantaneous.

Requirements/Wish List In our consideration of coding, as we mentioned we need the code to be uniquely decodable, and we would like the code to be instantaneous.

There are two other things we would like. One is a systematic way to design good codes. Given any number of symbols and any probabilities for these symbols, it would be nice to have a method for assigning codewords with good performance (i.e., small average bits per symbol). The second is a benchmark. It would be useful to know what is the smallest number of bits/symbol that could possibly be achieved with a uniquely decodable (and hopefully instantaneous) code, and it would nice to have if we had a method to achieve the best possible.

Thus, our requirements/wish list can be summarized as follows:

- Need a uniquely decodable code.

- Would like an instantaneous code.
- Would like a systematic design method.
- Would like a benchmark on the lowest possible bits/symbol that can be achieved.

Before addressing the first three items, we discuss the last.

8.6 Information and Entropy

Formalizing the notion of information and the limits to communication began in the late 1940's and continues to be an active area of research. This work forms another conceptual pillar of the information revolution and electrical engineering in general.

The formalization of "information" departs from the more usual notion relating to semantic meaning. Rather, the technical definition of the information provided by some event relates to the probability of occurrence of the event. Roughly speaking, the more surprise we experience upon observing a particular outcome, the more information provided by that outcome.

In particular, if the probability of an event is 1, then the event is certain. In this case, there is no surprise upon learning that the event occurred, and likewise we receive no information from knowledge of its occurrence (since we knew the event would occur). On the other hand, if the probability of an event is near 0, then the event is very unlikely and we would be very surprised to learn that the event occurred. In this case, we receive a great deal of information upon learning of its occurrence. In the intermediate case that the probability of occurrence is 1/2, both the occurrence and non-occurrence are equally likely. In this case, we receive exactly 1 bit of information upon learning the event occurred.

The amount of information we receive upon learning an event occurred is inversely related to the probability p of the event. To get other natural properties of information, it can be shown that the appropriate definition is

$$\begin{array}{c}\text{information gained upon learning} \\ \text{event of probability } p \text{ occurred}\end{array} \quad = \quad \log_2 \frac{1}{p} \ \text{ bits.}$$

In many applications there are a number of possible mutually exclusive outcomes denoted by symbols $s_1, s_2, \ldots, s_M$. For example, in an image processing application, we might have 256 symbols where $s_1, \ldots, s_{256}$ denote the possible gray levels of a particular pixel. For audio signals, the s_i could denote the amplitude level of a sample of the signal. In text applications, the s_i might denote the various letters, digits, punctuation, and other special characters that are used. In most applications including those just mentioned, instead of having just a single symbol, we have a stream or "source" of many symbols. For example, in one 512×512 image we have over 260,000 symbols, in one second of digital audio we have 44,100 symbols, and in text document we will typically have thousands

or millions of symbols. The motivation for introducing the s_i and talking about a source of symbols is to keep the development general enough to be applied to all these (and other) examples.

If someone is sending us a stream symbols from a source (such as those above), we generally have some uncertainty about the sequence of symbols we will be receiving. If not, there would no need for the transmitter to even bother sending us the data. A useful and common way to model this uncertainty is to assume that the data is coming randomly according to some probability distribution. The simplest model is to assume that each the symbols $s_1, \ldots, s_M$ have associated probabilities of occurrence $p_1, \ldots, p_M$, and we see a string of symbols drawn independently according to these probabilities. Since the p_i are probabilities, we have $0 \leq p_i \leq 1$ for all i. Also, we assume the only possible symbols in use are the $s_1, \ldots, s_M$, and so $p_1 + \cdots + p_M = 1$.

How much information does a source provide? Consider just one symbol from the source. The probability that we observe the symbol s_i is p_i, so that if we do indeed observe s_i then we get $\log_2 \frac{1}{p_i}$ bits of information. Therefore, the average number of bits of information we get based on observing one symbol is

$$p_1 \log_2 \frac{1}{p_1} + \cdots + p_M \log_2 \frac{1}{p_M} = \sum_{i=1}^{M} p_i \log_2 \frac{1}{p_i}$$

This is an important quantity called the entropy of the source and is denoted by H. Note that the entropy H should not be confused with the frequency response of a system $H(\omega)$ discussed in Chapter XX. These two areas developed somewhat independently and the use of $H(\cdot)$ for frequency response is very common and the use of H for entropy in information theory is universal.

Definition Given a source that outputs symbols $s_1, \ldots, s_M$ with probabilities $p_1, \ldots, p_M$, respectively, the *entropy* of the source, denoted H, is defined as

$$H = \sum_{i=1}^{M} p_i \log_2 \frac{1}{p_i}$$

.

The importance of the entropy of a source lies in its operational significance concerning coding the source. Since H represents the average number of bits of information per symbol from the source, we might expect that we need to use at least H bits per symbol to represent the source with a uniquely decodable code. This is in fact the case, and moreover, if we wish to code longer and longer strings of symbols, we can find codes whose performance (average number of bits per symbol) gets closer to H. This result is called the source coding theorem and was discovered by Shannon in 1948.

Source Coding Theorem
(i) The average number of bits/symbol of any uniquely decodable source must be greater than or equal to the entropy H of the source.

(ii) If the string of symbols is sufficiently large, there exists a uniquely decodable code for the source such that the average number of bits/symbol of the code is as close to H as desired.

This result addresses the fourth item on our wish list. It shows that the entropy of a source is the benchmark we are after. No code can do better than H bits/symbol on average, and if the string of symbols to be coded is long enough then there exist codes that can get us as close to optimal as we like. In the next section, we consider a particular type of code known as Huffman coding that gives us a systematic design method for finding good codes once we know the symbol probabilities. This addresses the first, second, and third items on our wish list.

8.7 Huffman Coding

Huffman coding is a simple and systematic way to design good variable-length codes given the probabilities of the symbols. The resulting code is both uniquely decodable and instantaneous (prefix-free). Huffman coding is used in many applications. For example, as we will see in Chapter XX, it is a component of standard image compression (JPEG), video compression (MPEG), and codes used in fax machines.

The Huffman coding algorithm can be summarized as follows:

1. Think of the p_i as the leaf nodes of a tree. In constructing a Huffman code by hand it's sometimes useful to sort the p_i in decreasing order.

2. Starting with the leaf nodes, construct a tree as follows. Repeatedly join two nodes with the smallest probabilities to form a new node with the sum of the probabilities just joined. Assign a 0 to one branch and a 1 to the other branch. In constructing Huffman codes by hand, it's often helpful to do this assignment in a systematic way, such as always assigning 0 to the branch on the same side.

3. The codeword for each symbol is given by the sequence of 0's and 1's starting *from* the root node and leading *to* the leaf node corresponding to the symbol.

The order of reading off the o's and 1's in constructing the codewords is important. If the order is reversed, the resulting code will generally not be instantaneous.

Example 8.1 (A Simple Huffman Code)
Consider a source with symbols s_1, s_2, s_3, s_4 with probabilities 1/2, 1/4, 1/8, 1/8, respectively. The Huffman code is constructed as shown in Figure 8.1.

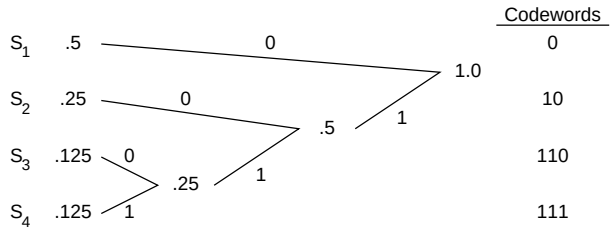

Figure 8.1: Construction of simple Huffman code with 4 symbols.

From the tree constructed, we can read off the codewords. For example, the codeword for s_2 is found by starting at the root node and following the branches labeled 1 then 0 to arrive at s_2. Therefore, the codeword for s_2 is 10.

The average number of bits per symbol for this code is

$$\begin{aligned} \text{average length} &= (1)\left(\frac{1}{2}\right) + (2)\left(\frac{1}{4}\right) + (3)\left(\frac{1}{8}\right) + (3)\left(\frac{1}{8}\right) \\ &= 1.75 \text{ bits/symbol} \end{aligned}$$

Note that a fixed length code would require 2 bits/symbol.

What is best we can do for the given source? The answer is provided by the entropy, which is given by

$$\begin{aligned} H &= \frac{1}{2}\log_2 2 + \frac{1}{4}\log_2 4 + \frac{1}{8}\log_2 8 + \frac{1}{8}\log_2 8 \\ &= \frac{1}{2} + \frac{1}{2} + \frac{3}{8} + \frac{3}{8} \\ &= 1.75 \end{aligned}$$

■

In the example above, the rate of the Huffman code is exactly the entropy, so that the Huffman code is the best possible. This is not always the case, as can be seen in the example below. However, it can be shown that that

$$H \leq \text{average length of Huffman code} \leq H + 1$$

.

Example 8.2 (A Slightly More Involved Huffman Code)

Consider a source with eight symbols $s_1, \ldots, s_8$ with probabilities 0.25, 0.21, 0.15, 0.14, 0.0625, 0.625, 0.625, 0.625, respectively. The Huffman code is constructed as shown in Figure 8.2.

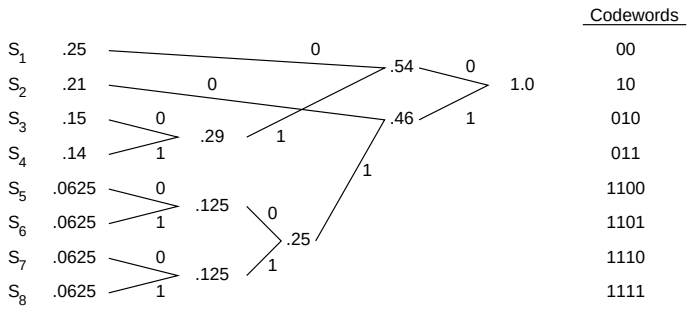

Figure 8.2: A slightly more involved Huffman code with 8 symbols.

Note that in this example, at the stage when we combined the node with probability 0.21 (corresponding to s_2) with the node with probability 0.25 (leading to s_5, s_6, s_7, s_8), we could have instead combined s_2 with s_1 since the node corresponding to s_1 also has probability 0.25. Whenever there is more than one choice, they are all acceptable and lead to valid Huffman codes. Of course, the specific codewords and even the lengths of the codewords may be different for the different choices. However, all will result in the same average codeword length. The average number of bits per symbol for this code is

$$\begin{aligned} \text{average length} &= (2)(0.25+0.21)+(3)(0.15+0.14) \\ &\quad +(4)(0.0625+0.0625+0.0625+0.0625) \\ &= 2.79 \text{ bits/symbol} \end{aligned}$$

Note that a fixed length code would require 3 bits/symbol.

For this source the entropy is given by

$$\begin{aligned} H &= 0.25\log_2\left(\frac{1}{0.25}\right)+0.21\log_2\left(\frac{1}{0.21}\right)+0.15\log_2\left(\frac{1}{0.15}\right) \\ &\quad +0.14\log_2\left(\frac{1}{0.14}\right)+(4)(0.0625)\log_2\left(\frac{1}{0.0625}\right) \\ &\approx 2.78 \end{aligned}$$

■

In this example, the average length of the Huffman code is close, but not equal, to the entropy of the source. In other cases, the Huffman code may not be quite as close, but as mentioned previously, the average length of a Huffman code will never differ from the entropy by more than one. However, the source coding theorem claims that we can find a code whose average length is arbitrarily close to the entropy. The next section shows how with a simple extension, we can use Huffman codes and to get an average length arbitrarily close to H.

8.8 Block Coding

Consider a very simple source with just two sybols s_1, s_2 with probabilities 0.9 and 0.1, respectively. The Huffman code for this source (shown in Figure 8.3) results in the codewords 0 and 1.

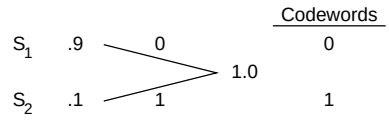

Figure 8.3: A trivial Huffman code with 2 symbols to motivate block coding.

Thus, the Huffman code is just a fixed length code and has average length of 1 bit/symbol. Of course, with just two symbols this is really the only reasonable choice for the codewords. On the other hand, the entropy of this source is

$$H = 0.9 \log_2 \left(\frac{1}{0.9} \right) + 0.1 \log_2 \left(\frac{1}{0.1} \right) \approx 0.47 \quad \text{bits/symbol}$$

In this case, the Huffman code does poorly compared with the entropy. This isn't surprising. Since there are only two symbols, we have no choice but to assign one bit to each symbol, even though the probabilities of the two symbols are quite different. We can reap more benefits from variable-length coding when there are many symbols to choose from in the design of codeword lengths.

A technique known as block coding is a way to better exploit the advantages of assigning different codewords lengths using variable length coding. The idea is to consider more than one symbol at a time and assign codewords to these "blocks" of symbols. The advantage comes not only from more flexibility in assigning codeword lengths, but also from the fact that the different probabilities of groups of symbols get magnified when we consider groups of symbols. Blocks containing low probability symbols have extremely small probability compared with blocks containing more commonly occurring symbols.

For example, using blocks of length 2 for the source above with two symbols, we have four possibilities we need to encode: $s_1 s_1$, $s_1 s_2$, $s_2 s_1$, and $s_2 s_2$. These pairs of symbols have probabilities 0.81, 0.09, 0.09, and 0.01, respectively. The Huffman code for blocks of two symbols is designed as shown in Figure 8.4.

The average length for encoding *two* symbols is

$$1(0.81) + 2(0.09) + 3(0.09 + 0.01) = 1.29.$$

Therefore, the average number of bits per *original* symbol is $1.29/2 = 0.645$ bits/symbol.

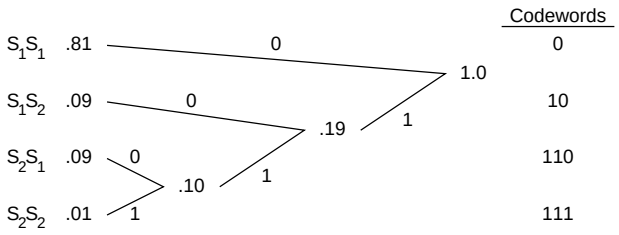

Figure 8.4: An example of block coding using blocks os length 2 for the source in Figure 8.3.

Using blocks of length 2 helps significantly by reducing the average codeword length from 1 bit/symbol to 0.645 bits/symbol. We could get even smaller average codeword length with longer blocks. In fact, as the length of the block gets larger, the average codeword length gets closer to the entropy. We can get as close to the entropy as desired by taking sufficiently large blocks. However, there are practical considerations that prevent the use of very large blocks. These and some other issues are discussed in the next section.

8.9 Comments on Huffman Coding

Huffman codes are uniquely decodable and instantaneous codes. Recall that an instantaneous code is equivalent to the prefix-free property. Both unique-decodability and the prefix-free property can be seen from the fact that the codewords of a Huffman code are the leaves of a binary tree. If we see a string of bits and follow from the root node down the tree taking branches according to the bits seen. Then we will eventually reach a leaf node corresponding to the correct symbol. Once the bits for a symbol are completed then we can immediately decode (so the code is instantaneous), and we start again from the root node to continue the decoding. We can also see that there is only one way to decode a given string of symbols (so the code is uniquely decodable).

A Huffman tree is not wasteful in the sense that all the leaf nodes correspond to symbols. If there were empty leaf nodes, then we could remove these and shorten certain codewords and still have a tree code. In fact, it can be shown that for a given block size, Huffman codes are in fact optimal.

However, the average length of the Huffman code may not equal the entropy. But the Huffman code does satisfy

$$H \leq \text{average length of Huffman code} \leq H + 1$$

To approach the entropy, we can use block coding. As the block size gets larger, the average length of the corresponding Huffman code approaches the entropy. In fact using a block size k, it can be shown that

$$H \leq \text{average length per original symbol} \leq H + \frac{1}{k}$$

Unfortunately, for practical reasons the block size cant be taken to be too large. In fact, if the source has M symbols, then with blocks of length k there are M^k strings of symbols to consider. This grows exponentially in the block size k, and even for modest M and k this can be impractically large. Note that the encoder and decoder both need the codewords. The encoder has to form the Huffman tree using the probabilities, and this either has to be sent as overhead to the decoder, or the decoder needs the same probabilities to construct the Huffman tree itself.

There are two other major disadvantages/limitations of Huffman codes. One involves the assumptions on the source. We assumed that the symbols are randomly drawn (independent and identically distributed) and the probabilities of the symbols are known. Both of these assumptions are unrealistic in many applications. For example, in image processing, suppose we try to apply Huffman coding directly where the symbols are taken to be the gray levels of the pixels. Do we know the probability that a given pixel will take on each gray level? Surely, this depends greatly on the class of images under consideration, and even for very restricted applications this knowledge is hard to come by. The assumption that the gray levels are independent and identically distributed is perhaps even worse. In most images encountered there is a good deal of correlation in the gray levels of nearby pixels. Images with independent and identically distributed pixels look like pure noise. Basic Huffman coding can't fully exploit this additional structure.

The second main disadvantage with Huffman coding is that any errors in the coded sequence of bits will tend to propagate when decoding. This is an issue for most variable length codes since, unlike fixed length codes, the boundaries between codewords is not known in advance, but rather is "discovered" during the decoding process. If there is an error, then unless the incorrectly decoded sequence of bits happens to end on the correct codeword boundary, the error will propagate into the next codeword.

Thus, although Huffman coding has a number of nice properties and is used in a number of applications (some of which we will discuss in the next chapter), there are significant limitations as well. The impracticality of using large block sizes together with the assumption that symbols are independent and identically distributed with known probabilities, forcefully show that Huffman coding far from a panacea for data compression. In fact this is widely studied area with a number of general techniques as well as methods tailored to specific types of data. The next chapter discusses some of these techniques.

There are several mechanisms that can be used to prevent error propagation. Some of these techniques have limited use beyond preventing error propagation and we will not discuss these. But one approach that can be used has much the much broader purpose protecting the data itself against errors (whether or not the errors propagate). These techniques are designed to detect and/or correct error, and form one topic in the very important area of information protection. This is the subject of Chapter XX.