store an image with each of the following display sizes:
(i) 1024 x 768,
(ii) 1280 x 1024?

Derive the time to transmit an image with each type of display assuming a bit rate of
(i) 56 kbps,
(ii) 1.5 Mbps.

2.25 With the aid of a diagram, explain how a digital image produced by a scanner or digital camera is captured and stored within the memory of a computer.

2.26 With the aid of a diagram, explain how a color image is captured within a camera or scanner using each of the following methods:
(i) single image sensor,
(ii) a single image sensor with filters,
(iii) three separate image sensors. Include in your explanations the terms "photosites" and "CCDs" and the role of the readout register.

Section 2.5

2.27 With the aid of a diagram, explain the principle of operation of a PCM speech codec. Include in your diagram the operation of the compressor in the encoder and the expander in the decoder. Use for example purposes 5 bits per sample.

2.28 Identify the main features of the MIDI standard and its associated messages.

Section 2.6

2.29 With the aid of a diagram, explain the principles of interlaced of scanning as used in most TV broadcast applications. Include in your explanation the meaning of the terms "field", "odd scan lines", and "even scan lines". Show the number of scan lines per field with
(i) a 525-line system and
(ii) a 625-line system. Why do computer monitors not use interlaced scanning?

2.30 State and explain the three main properties of a color source that the eye makes of. Hence explain the meaning of the terms "luminance", "chrominance", and "color difference" and how the magnitude of each primary color present in the source is derived from these.

2.31 Why is the chrominance signal transmitted in the form of two color different signals? Identify the color difference signals associated with the NTSC and PAL systems.

2.32 State the meaning of the term "composite video signal" and, with the aid of a diagram, describe how the two color difference signals are transmitted within the same frequency band as that used for the luminance signal.

2.33 Explain why, for digital TV transmission, the three digitized signals used are the luminance and two color difference signals rather than the RGB signals. Why are a number of different digitization formats used?

2.34 With the aid of diagrams, describe the following digitization formats:
(i) 4:2:2,
(ii) 4:2:0,
(iii) SIF,
(iv) CIF,
(v) QCIF,
(vi) S-QCIF.

For each format, state the temporal resolution and the sampling rate used for the luminance and the two color difference signals. Give an example application of each format.

2.35 Derive the bit rate that results from the digitization of a 525-line and a 625-line system using the 4:2:0 digitization format and interlaced scanning. Hence derive the amount of memory required to store a 2-hour movie/ video.

2.36 Explain why modifications to the received (broadcast) TV signal have to be made if the signal is to be displayed in a window of a computer monitor. Hence assuming the SIF format, derive the spatial resolution required with
(i) a 525-line and
(ii) a 625-line system.

Text and image compression

3.1 Introduction

In the previous chapter we described the way the different types of media used in multimedia applications – text, fax, images, speech, audio, and video – are represented in a digital form. We derived the memory and bandwidth requirements for each type and, as we concluded in Section 2.7, in most cases, the bandwidths derived were greater than those that are available with the communication networks over which the related services are provided. In addition, when using a public network in which call charges are based on the duration of a call, considerable cost savings can be made if the volume of information to be transmitted is reduced.

In almost all multimedia applications, therefore, a technique known as **compression** is first applied to the source information prior to its transmission. This is done either to reduce the volume of information to be transmitted – text, fax, and images – or to reduce the bandwidth that is required for its transmission – speech, audio, and video. In this chapter we shall consider a selection of the compression algorithms which are used with text, fax, and images and, in Chapter 4, we shall describe a selection of the compression algorithms that are used with audio and video.

3.2 Compression principles

Before we describe some of the no of compression algorithms in widespread use, it will be helpful if we first build up an understanding of the principles on which they are based. We shall discuss the under the headings:

- source encoders and destination decoders,
- lossless and lossy compression,
- entropy encoding,
- source encoding.

3.2.1 Source encoders and destination decoders

As have just indicated, prior to transmitting the source information relating to a particular multimedia application, a compression algorithm is applied to it. This implies that in order for the destination to reproduce the original source information – or, in some instances, a nearly exact copy of it – a matching decompression algorithm must be applied to it. The application of the compression algorithm is the main function carried out by the **source encoder** and the decompression algorithm is carried out by the **destination decoder**.

In applications which involve two computers communicating with each other, the time required to perform the compression and decompression algorithms is not always critical. So both algorithms are normally implemented in software within the two computers. The general scheme is shown in part (a) of Figure 3.1 and an example application which uses this approach is the compression of text and/or image files. In other applications, however, the time required to perform the compression and decompression algorithms in software is not acceptable and instead the two algorithms must be performed by special processors in separate units as shown in part (b) of the figure. Example applications which use this approach are those which involve speech, audio, and video.

3.2.2 Lossless and lossy compression

Compression algorithms can be classified as being either lossless or lossy. In the case of a **lossless compression algorithm** the aim is to reduce the amount of source information to be transmitted in such a way that, when the compressed information is decompressed, there is no loss of information. Lossless compression is said, therefore, to be **reversible**. An example application of lossless compression is for the transfer over a network of a text file since, in such applications, it is normally imperative that no part of the source information is lost during either the compression or decompression operations.

In contrast, the aim of **lossy compression algorithms**, is normally not to reproduce an exact copy of the source information after decompression but

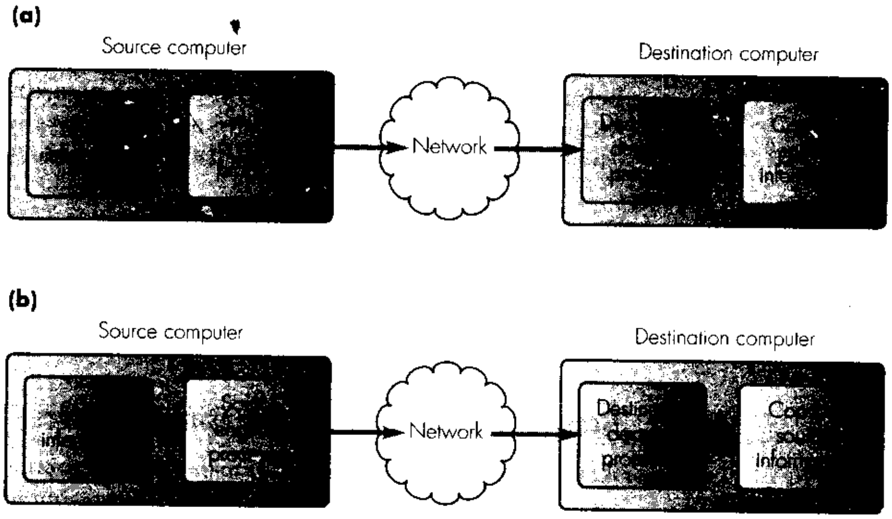


Figure 3.1 Source encoder/destination decoder alternatives: (a) software only; (b) special processors/hardware.

rather a version of it which is perceived by the recipient as a true copy. In general, with such algorithms the higher the level of compression being applied to the source information the more approximate the received version becomes. Example applications of lossy compression are for the transfer of digitized images and audio and video streams. In such cases, the sensitivity of the human eye or ear is such that any fine details that may be missing from the original source signal after decompression are not detectable.

3.2.3 Entropy encoding

Entropy encoding is lossless and independent of the type of information that is being compressed. It is concerned solely with how the information is represented. We shall describe two examples which are in widespread use in compression algorithms in order to illustrate the principles involved. In some applications they are used separately while in others they are used together.

, lossless
, not depend → type of info
, concerned only with → how the info is represented

Run Length coding

Run-length encoding

Typical applications of this type of encoding are when the source information comprises long substrings of the same character or binary digit. Instead of transmitting the source string in the form of independent codewords or bits, it is transmitted in the form of a different set of codewords which indicate not only the particular character or bit being transmitted but also an indication of the number of characters/bits in the substring. Then, providing the destination knows the set of codewords being used, it simply interprets each codeword received and outputs the appropriate number of characters or bits.

, app → [illegible] → long substrings of same char or binary digit.

independent codeword or bits | diff set of codewords → indicate char also no. of char in the substring

interprets the codeword o/p the appropriate no. of char.

For example, in an application that involves the transmission of long strings of binary bits that comprise a limited number of substrings, each substring can be assigned a separate codeword. The total bit string is then transmitted in the form of a string of codewords selected from the codeword set. An example application which uses this technique is for the transmission of the binary strings produced by the scanner in a facsimile machine. In many instances – for example when scanning typed documents – the scanner produces long substrings of either binary 0s or 1s. Instead of transmitting these directly, they are sent in the form of a string of codewords, each indicating both the bit – 0 or 1 – and the number of bits in the substring. For example, if the output of the scanner was:

eg: FAX the scanner produces long substrings of 0's & 1's

0000000111111111110000011... eg:

they are sent in the form of a string of codewords (each indicating 0's & 1's & no of bits in the substring)

this could be represented as: 0,7 1,10 0,5 1,2 ... . Alternatively, since only the two binary digits 0 and 1 are involved, if we ensure the first substring always comprises binary 0s, then the string could be represented as 7, 10, 5, 2 ... . To send this in a digital form, the individual decimal digits would be sent in their binary form and, assuming a fixed number of bits per codeword, the number of bits per codeword would be determined by the largest possible substring. We shall describe an application that uses this approach in Section 3.4.3 when we describe the compression of digitized documents.

Statistical encoding

(1) Many apps eg. ASCII codewords for tx of strings of char → tx of strings of char

Many applications use a set of codewords ~~to transmit the source information~~. For example, as we described earlier in Section 2.3.1, a set of ASCII codewords are often used for the transmission of strings of characters. Normally, all the codewords in the set comprise a fixed number of binary bits, for example 7 bits in the case of ASCII. (2) In many applications, however, the symbols – and hence codewords – that are present in the source information do not occur with the same frequency of occurrence; that is, with equal probability. (3) For example, in a string of text, the character A may occur more frequently than, say, the character P which occurs more frequently than the character Z, and so on. Statistical encoding exploits this property by using a set of variable-length codewords with the shortest codewords used to represent the most frequently occurring symbols.

(4) In practice, the use of variable-length codewords is not quite as straightforward as it first appears. Clearly, as with run-length encoding, the destination must know the set of codewords being used by the source. With variable-length codewords, however, in (5) order for the decoding operation to be carried out correctly, it is necessary to ensure that a shorter codeword in the set does not form the start of a longer codeword otherwise the decoder will interpret the string on the wrong codeword boundaries. A codeword set that avoids this happening is said to possess the **prefix property** and an example of an encoding scheme that generates codewords that have this property

(6) (7)

is the **Huffman encoding algorithm** which we shall describe in Section 3.3.1.

The theoretical minimum average number of bits that are required to transmit a particular source stream is known as the **entropy** of the source and can be computed using a formula attributed to Shannon:

min avg no. of bits reqd to tx a particular source stream – Entropy (8)

$$\text{Entropy, } H = -\sum_{i=1}^{n} P_i \log_2 P_i$$

Entropy, $H = -\sum_{i=1}^{n} P_i$

where n is the number of different symbols in the source stream and P_i is the probability of occurrence of symbol i. Hence the efficiency of a particular encoding scheme is often computed as a ratio of the entropy of the source to the average number of bits per codeword that are required with the scheme. The latter is computed using the formula:

(9) Efficiency of an encoding scheme = $\frac{\text{entropy}}{\text{avg no of bits reqd with code word}}$

$$\text{Average number of bits per codeword} = \sum_{i=1}^{n} N_i P_i$$

(10)

Example 3.1

A statistical encoding algorithm is being [illegible]
sion of a large number [illegible]
[illegible]
ferent [illegible]
frequency of occurrence of [illegible]
respectively. If the [illegible]
lowing set of codewords:

M = 10, F = 11, Y = [illegible]

compute:

(i) the average number [illegible]

(ii) the entropy of the source,

(iii) the minimum number of bits required assuming fixed-length codewords.

Answer:

(i) Average number of bits per codeword

$$= \sum_{i=1}^{6} N_i P_i = (2(2 \times 0.25) + 4(3 \times 0.125))$$

$$= 2 \times 0.5 + 4 \times 0.375 = 2.5$$

3.1 Continued

(ii) Entropy of source

$$= \sum_{i=1}^{6} P_i \log_2 P_i = -(2(0.25\log_2 0.25) + 4(0.125\log_2 0.125))$$

$$= 1 + 1.5 = 2.5$$

(iii) Since there are 6 different characters, using fixed-length codewords would require a minimum of 3 bits (8 combinations).

3.2.4 Source encoding

(1)

Source encoding exploits a particular property of the source information in order to produce an alternative form of representation that is either a compressed version of the original form or is more amenable to the application of compression. Again, we shall describe two examples in widespread use in order to illustrate the principles involved.

particular property of source info

alt. form of rep

compressed version of original | more amenable to the app of compression

Differential encoding

apps

amp of a value/signal (large range)

diff in amp b/w successive values/signals is small

Differential encoding is used extensively in applications where the amplitude of a value or signal covers a large range but the difference in amplitude between successive values/signals is relatively small. To exploit this property of the source information, instead of using a set of relatively large codewords to represent the amplitude of each value/signal, a set of smaller codewords can be used each of which indicates only the difference in amplitude between the current value/signal being encoded and the immediately preceding value/signal. For example, if the digitization of an analog signal requires, say, 12 bits to obtain the required dynamic range but the maximum difference in amplitude between successive samples of the signal requires only 3 bits, then by using only the difference values a saving of 75% on transmission bandwidth can be obtained.

to rep [illegible]

eg.

In practice, differential encoding can be either lossless or lossy and depends on the number of bits used to encode the difference values. If the number of bits used is sufficient to cater for the maximum difference value then it is lossless. If this is not the case, then on those occasions when the difference value exceeds the maximum number of bits being used, temporary loss of information will result.

Transform encoding

(2)

As the name implies, transform encoding involves transforming the source information from one form into another, the other form lending itself

2

more readily to the application of compression. In general, there is no loss of information associated with the transformation operation and this technique is used in a number of applications involving both images and video. For example, as we saw in Section 2.4.3, the digitization of a continuous-tone monochromatic image produces a two-dimensional matrix of pixel values each of which represents the level of gray in a particular position of the image. As we go from one position in the matrix to the next, the magnitude of each pixel value may vary. Hence, as we scan across a set of pixel locations, the rate of change in magnitude will vary from zero, if all the pixel values remain the same, to a low rate of change if, say, one half is different from the next half, through to a high rate of change if each pixel magnitude changes from one location to the next. Some examples are shown in Figure 3.2(a).

eg

cont. monotone
monochromatic
image
↓
2D matrix of
pixel values
|
each pixel
rep. a gray value
rate of magnitude of
each pixel may vary

The rate of change in magnitude as one traverses the matrix gives rise to a term known as **spatial frequency** and, for any particular image, there will be a mix of different spatial frequencies whose amplitudes are determined by the related changes in magnitude of the pixels. This is true, of course, if we scan the matrix in either the horizontal or the vertical direction and this, in turn, gives rise to the terms **horizontal** and **vertical frequency components** of the image. In practice, the human eye is less sensitive to the higher spatial frequency components associated with an image than the lower frequency components. Moreover, if the amplitude of the higher frequency components falls below a certain amplitude threshold, they will not be detected by the eye. Hence in terms of compression, if we can transform the original spatial form of representation into an equivalent representation involving spatial frequency components, then we can more readily identify and eliminate those higher frequency components which the eye cannot detect thereby reducing the volume of information to be transmitted without degrading the perceived quality of the original image.

Human eye

less sensitive HF | compared to LF

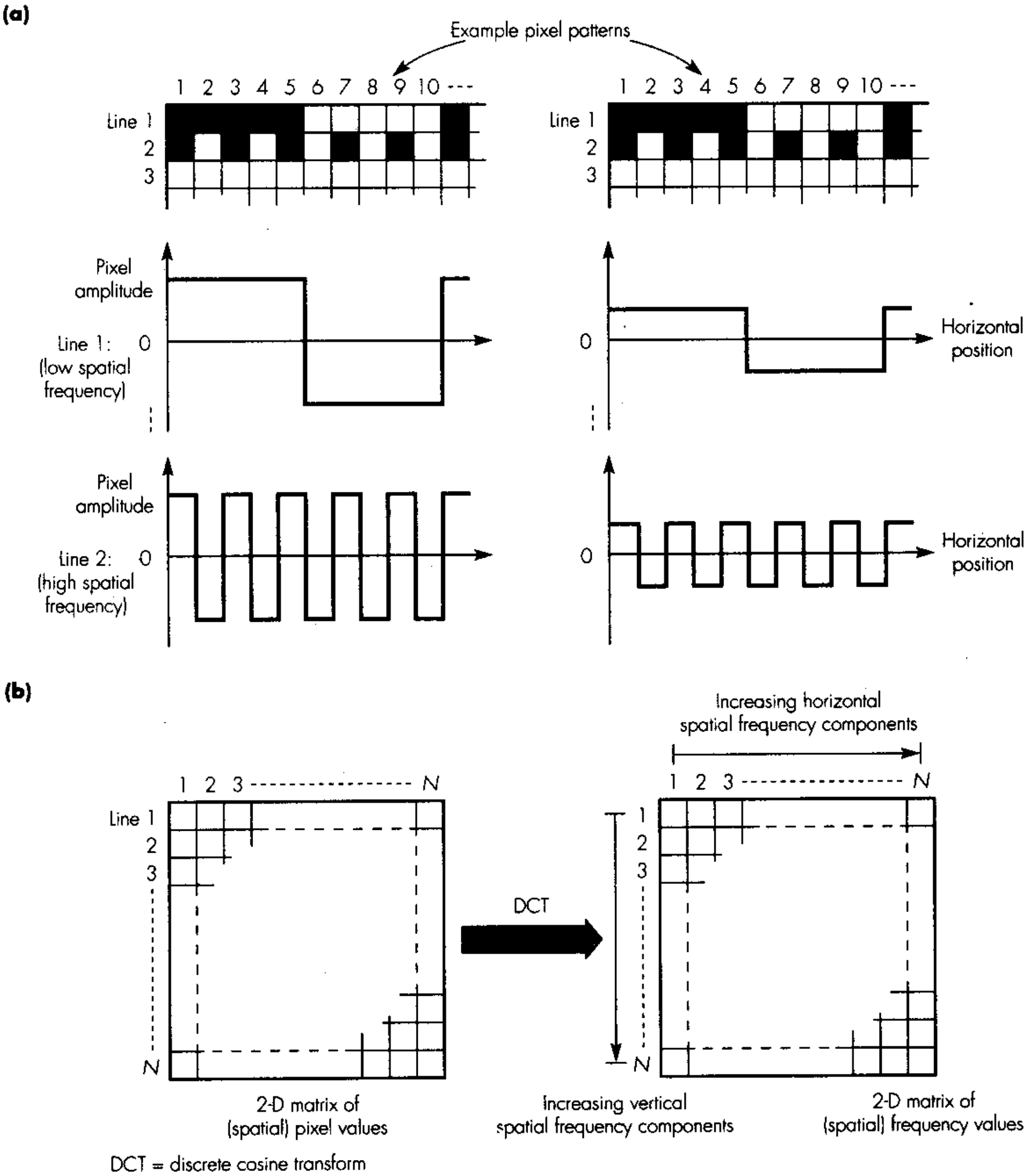
if amp of HF falls below certain amplitude
they will not be detected by the eye

compression
if we can eliminate those HF that eye cannot detect.
↓
reduction of info w/o degrading quality

The transformation of a two-dimensional matrix of pixel values into an equivalent matrix of spatial frequency components can be carried out using a mathematical technique known as the **discrete cosine transform (DCT)**. The transformation operation itself is lossless – apart from some small rounding errors in the mathematics – but, once the equivalent matrix of spatial frequency components – known as coefficients – has been derived, then any frequency components in the matrix whose amplitude is less than a defined threshold can be dropped. It is only at this point that the operation becomes lossy. The basic principle behind the DCT is as shown in Figure 3.2(b) and we shall describe it in more detail in Section 3.4.5 when we discuss the topic of image compression.

2D matrix of pixel values $\xrightarrow[\text{(lossless)}]{\text{DCT}}$ matrix of spatial freq comp (coeff)

once this is obtained then any freq components in the matrix whose amp is less than a defined threshold can be dropped (lossy)



DCT = discrete cosine transform

Figure 3.2 Transform coding: (a) example pixel patterns; (b) DCT transform principles.

3.3 Text compression

As we saw in Section 2.3, the three different types of text – unformatted, formatted, and hypertext – are all represented as strings of characters selected from a defined set. The strings comprise alphanumeric characters which are interspersed with additional control characters. The different types of text use and interpret the latter in different ways. As we can deduce from this, any compression algorithm associated with text must be lossless since the loss of just a single character could modify the meaning of a complete string. In general, therefore, we are restricted to the use of entropy encoding and, in practice, statistical encoding methods.

text
lossless compression
↓
entropy encoding
(Statistical)

Essentially, there are two types of statistical encoding methods which are used with text: one which uses single characters as the basis of deriving an optimum set of codewords and the other which uses variable-length strings of characters. Two examples of the former are the Huffman and arithmetic coding algorithms and an example of the latter is the **Lempel-Ziv (LZ) algorithm**. We shall describe the principles of each of these algorithms in this section.

2 types
Huff/AC
(1) uses single char. to derive optimum set of code words
(2) variable length string of characters
(LZ)

There are two types of coding used for text. The first is intended for applications in which the text to be compressed has known characteristics in terms of the characters used and their relative frequencies of occurrence. Using this information, instead of using fixed-length codewords, an optimum set of variable-length codewords is derived with the shortest codewords used to represent the most frequently occurring characters. The resulting set of codewords are then used for all subsequent transfers involving this particular type of text. This approach is known as static coding.

~~text with~~
2 types of coding
(1) when text has known chars & their relative freq of ocurence
↓
set of variable length codeword is derived
↓
shortest codeword used to rep. most freq. ocuring chr.

The second type is intended for more general applications in which the type of text being transferred may vary from one transfer to another. In this case the optimum set of codewords is also likely to vary from one transfer to another. To allow for this possibility, the codeword set that is used to transfer a particular text string is derived as the transfer takes place. This is done by building up knowledge of both the characters that are present in the text and their relative frequency of occurrence dynamically as the characters are being transmitted. Hence the codewords used change as a transfer takes place, but in such a way that the receiver is able to dynamically compute the same set of codewords that are being used at each point during a transfer. This approach is known as **dynamic** or **adaptive coding** and, since each uses a different algorithm to derive the codeword set, we shall describe each separately.

(2) type of text being transfered may vary with every transfer
↓
the optimum set of codeword also varies 4m one transfer to another

3.3.1 Static Huffman coding

With static Huffman coding the character string to be transmitted is first analyzed and the character types and their relative frequency determined. The coding operation involves creating an unbalanced tree with some branches (and hence codewords, in practice) shorter than others. The degree of imbal-

> codeword ~~string~~ set used to define a particular text string is derived as transfer takes place → done by building up knowledge of both the chars. that are present in the text & their rel. freq. of occurrence of [illegible] chr.

ance is a function of the relative frequency of occurrence of the characters: the larger the spread, the more unbalanced is the tree. The resulting tree is known as the **Huffman code tree.**

A Huffman (code) tree is a **binary tree** with branches assigned the value 0 or 1. The base of the tree, normally the geometric top in practice, is known as the **root node** and the point at which a branch divides, a **branch node**. The termination point of a branch is known as a **leaf node** to which the symbols being encoded are assigned. An example of a Huffman code tree is shown in Figure 3.3(a). This corresponds to the string of characters AAAABBCD.

As each branch divides, a binary value of 0 or 1 is assigned to each new branch: a binary 0 for the left branch and a binary 1 for the right branch. The codewords used for each character (shown in the leaf nodes) are determined by tracing the path *from* the root node out to each leaf and forming a

AAAABB CD

A4 4 4 } 8
B2 2 } 4
C1 1 } 2
D1 0

A = 1
B = 01
C : 001
D = 000

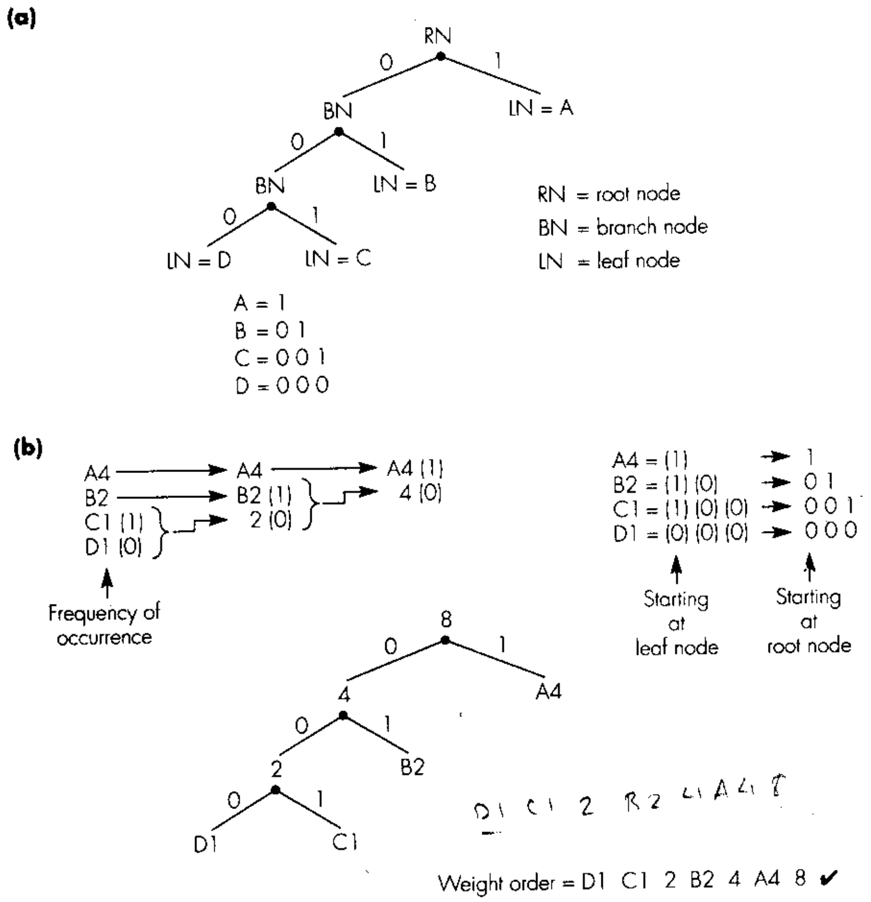


Figure 3.3 Huffman code tree construction: (a) final tree with codes; (b) tree derivation.

string of the binary values associated with each branch traced. We can deduce from the set of codes associated with this tree that it would take

$$4 \times 1 + 2 \times 2 + 1 \times 3 + 1 \times 3 = 14 \text{ bits}$$

to transmit the complete string AAAABBCD.

To illustrate how the Huffman code tree in Figure 3.3(a) is determined, we must add information concerning the frequency of occurrence of each character. Figure 3.3(b) shows the characters listed in a column in decreasing (weight) order. We derive the tree as follows.

The first two leaf nodes at the base of the list – C1 and D1 – are assigned to the (1) and (0) branches respectively of a branch node. The two leaf nodes are then replaced by a branch node whose weight is the sum of the weights of the two leaf nodes; that is, two. A new column is then formed containing the new branch node combined with the remaining nodes from the first column, again arranged in their correct weight order. This procedure is repeated until only two nodes remain.

To derive the resulting codewords for each character, we start with the character in the first column and then proceed to list the branch numbers – 0 or 1 – as they are encountered. Thus for character A the first (and only) branch number is (1) in the last column while for C the first is (1) then (0) at branch node 2 and finally (0) at branch node 4. The actual codewords, however, start at the root and not the leaf node hence they are the reverse of these bit sequences. The Huffman tree can then be readily constructed from the set of codewords.

We check that this is the optimum tree – and hence set of codewords – by listing the resulting weights of all the leaf and branch nodes in the tree starting with the smallest weight and proceeding from left to right and from bottom to top. The codewords are optimum if the resulting list increments in weight order.

Example 3.2

A series of messages is to be transferred between two computers over a PSTN. The messages comprise just the characters A through H. Analysis has shown that the probability (relative frequency of occurrence) of each character is as follows:

A and B = 0.25, C and D = 0.14, E, F, G, and H = 0.055

(a) Use Shannon's formula to derive the minimum average number of bits per character.

(b) Use Huffman coding to derive a codeword set and prove this is the minimum set by constructing the corresponding Huffman code tree.

3.2 Continued

(c) Derive the average number of bits per codeword for your codeword set and compare this with:

(i) the entropy of the messages (Shannon's value),

(ii) fixed-length binary codewords,

(iii) 7-bit ASCII codewords.

Answer:

(a) Shannon's formula states:

$$\text{Entropy, } H = -\sum_{i=1}^{n} P_i \log_2 P_i \text{ bits per codeword}$$

Therefore:

$$H = -(2(0.25 \log_2 0.25) + 2(0.14 \log_2 0.14) + 4(0.055 \log_2 0.055))$$
$$= 1 + 0.794 + 0.921 = 2.175 \text{ bits per codeword}$$

(b) The derivation of the codeword set using Huffman coding is shown in Figure 3.4(a). The characters are first listed in weight order and the two characters at the bottom of the list are assigned to the (1) and (0) branches. Note that in this case, however, when the two nodes are combined, the weight of the resulting branch node (0.11) is greater than the weight of the two characters E and F (0.055). Hence the branch node is inserted into the second list higher than both of these. The same procedure then repeats until there are only two entries in the list remaining.

The Huffman code tree corresponding to the derived set of codewords is given in Figure 3.4(b) and, as we can see, this is the optimum tree since all leaf and branch nodes increment in numerical order.

(c) Average number of bits per codeword using Huffman coding is:

$$2\,(2 \times 0.25) + 2(3 \times 0.14) + 4(4 \times 0.055) = 2.72 \text{ bits per codeword}$$

which is 99.8% of the Shannon value.

Using fixed-length binary codewords:

There are 8 characters – A through H – and hence 3 bits per codeword is sufficient which is 90.7% of the Huffman value.

Using 7-bit ASCII codewords:

7 bits per codeword
which is 38.86% of the Huffman value.

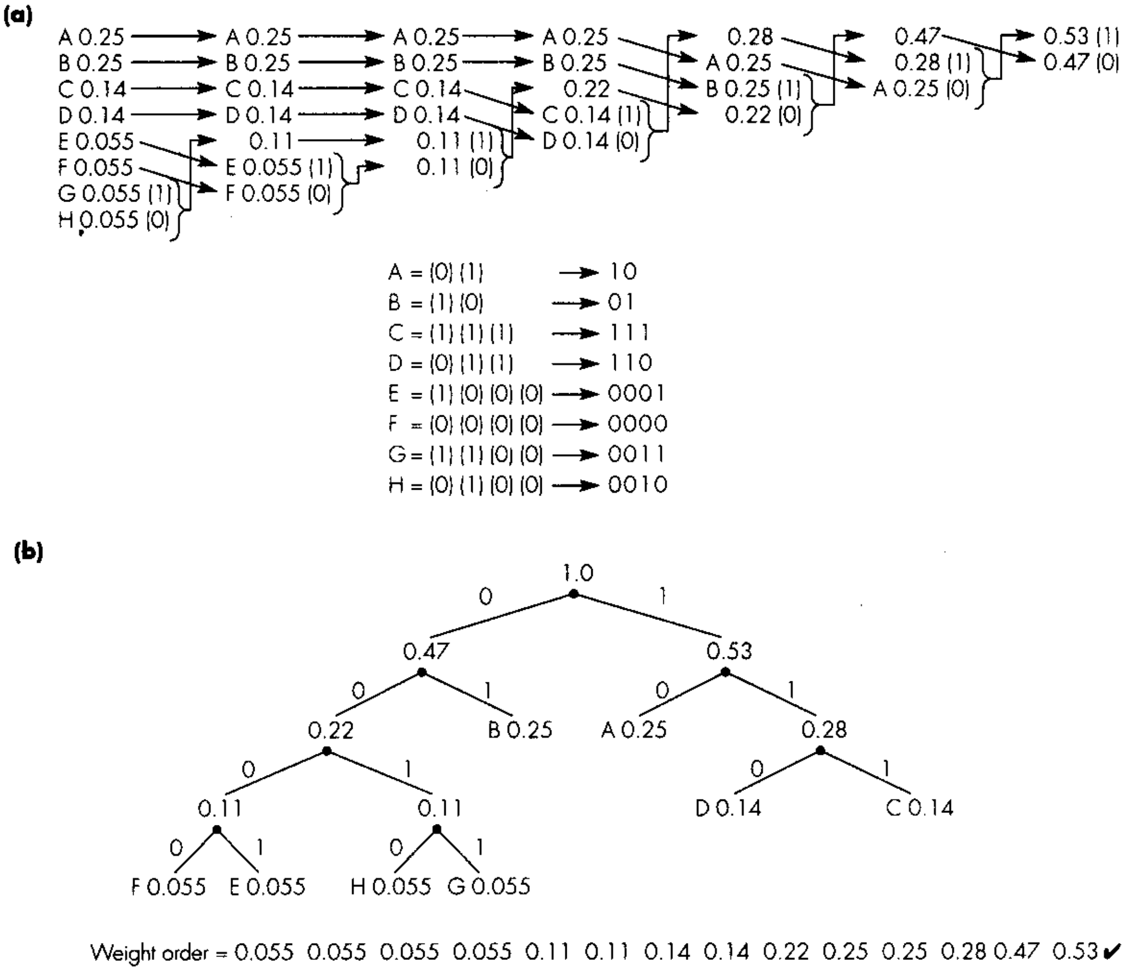


Figure 3.4 Huffman encoding example: (a) codeword generation; (b) Huffman code tree.

Since each character in its encoded form has a variable number of bits, the received bitstream must be interpreted (decoded) in a bit-oriented way rather than on fixed 7/8 bit boundaries. Because of the order in which bits are assigned during the encoding procedure, however, Huffman codewords have the unique property that a shorter codeword will never form the start of a longer codeword. If, say, 011 is a valid codeword, then there cannot be any longer codewords starting with this sequence. We can confirm this by considering the codes derived in the earlier examples in Figures 3.3 and 3.4.

This property, known as the **prefix property**, means that the received bitstream can be decoded simply by carrying out a recursive search bit by

prefix property
- each char → encoded → variable no of bits
rx bitstream → decoded in a bit oriented way rather then a fixed 7/8 bit boundary.

- Huffman
unique prop → shorter codewords will never form the start of a longer codeword starting with the sequence

rx bitstream can be decoded simply by carrying out a recursive search bit by bit
[illegible]

bit until each valid codeword is found. A flowchart of a suitable decoding algorithm is given in Figure 3.5(a). The algorithm assumes a table of codewords is available at the receiver and this also holds the corresponding ASCII codeword. The received bit stream is held in the variable BIT-STREAM and the variable CODEWORD is used to hold the bits in each codeword while it is being constructed. As we can deduce from the flowchart, once a codeword is identified the corresponding ASCII codeword is written into the variable RECEIVE_BUFFER. The procedure repeats until all the bits in the received string have been processed. An example of a decoded string corresponding to the codeword set derived in Figure 3.3 is given in Figure 3.5(b).

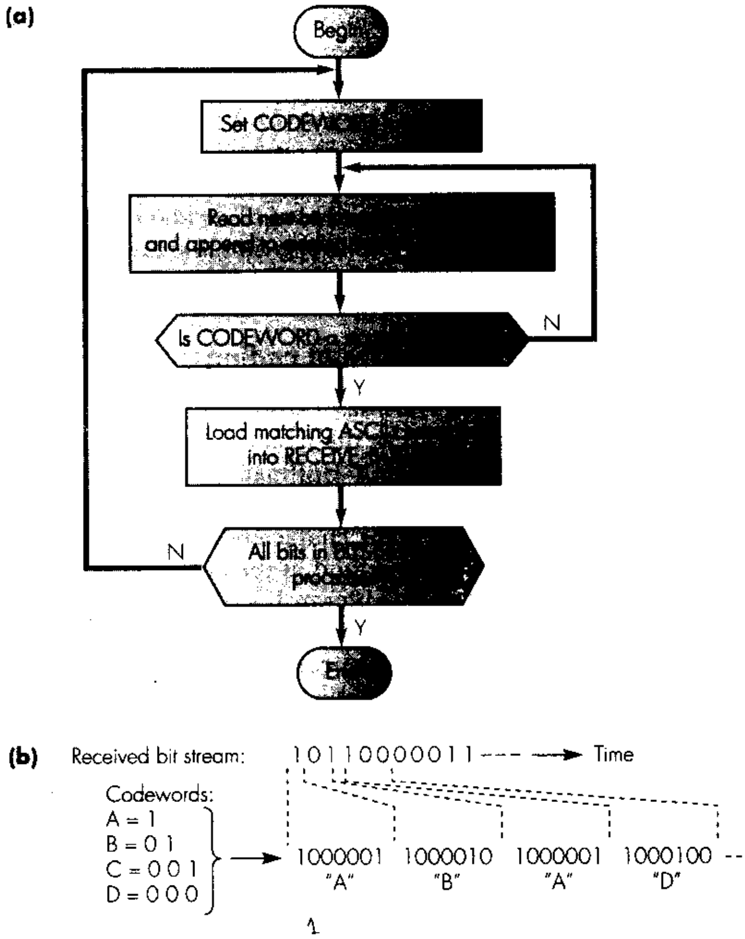


Figure 3.5 Decoding of a received bitstream assuming codewords derived in Figure 3.3: (a) decoding algorithm; (b) example.

As the Huffman code tree (and hence codewords) varies for different sets of characters being transmitted, for the receiver to perform the decoding operation it must know the codewords relating to the data being transmitted. This can be done in two ways. Either the codewords relating to the next set of data are sent before the data is transmitted, or the receiver knows in advance what codewords are being used.

The first approach leads to a form of adaptive compression since the codewords can be changed to suit the type of data being transmitted. The disadvantage is the overhead of having to send the new set of codewords (and corresponding characters) whenever a new type of data is to be sent. An alternative is for the receiver to have one or more different sets of codewords and for the sender to indicate to the receiver (through an agreed message) which codeword set to use for the next set of data.

For example, since a common requirement is to send text files generated by a word processor (and hence containing normal textual information), detailed statistical analyses have been carried out into the frequency of occurrence of the characters in the English alphabet in normal written text. This information has been used to construct the Huffman code tree for the alphabet. If this type of data is being sent, the transmitter and receiver automatically use this set of codewords. Other common data sets have been analyzed in a similar way and, for further examples, you may wish to consult the bibliography at the end of the book.

3.3.2 Dynamic Huffman coding

The basic Huffman coding method requires both the transmitter and the receiver to know the table of codewords relating to the data being transmitted. With dynamic Huffman coding, however, the transmitter (encoder) and receiver (decoder) build the Huffman tree – and hence codeword table – dynamically as the characters are being transmitted/received.

With this method, if the character to be transmitted is currently present in the tree its codeword is determined and sent in the normal way. If the character is not present – that is, it is its first occurrence – the character is transmitted in its uncompressed form. The encoder updates its Huffman tree either by incrementing the frequency of occurrence of the transmitted character or by introducing the new character into the tree.

Each transmitted codeword is encoded in such a way that the receiver, in addition to being able to determine the character that is received, can also carry out the same modifications to its own copy of the tree so that it can interpret the next codeword received according to the new updated tree structure.

To describe the details of the method, assume that the data (file) to be transmitted starts with the following character string:

This is simple ...

The steps taken by the transmitter are shown in Figure 3.6(a–g).

Both transmitter and receiver start with a tree that comprises the root node and a single **empty leaf node** – a leaf node with a zero frequency of occurrence – assigned to its 0-branch. There is always just one empty leaf node in the tree and its position – and codeword – varies as the tree is being constructed. It is represented in Figure 3.6 as e0.

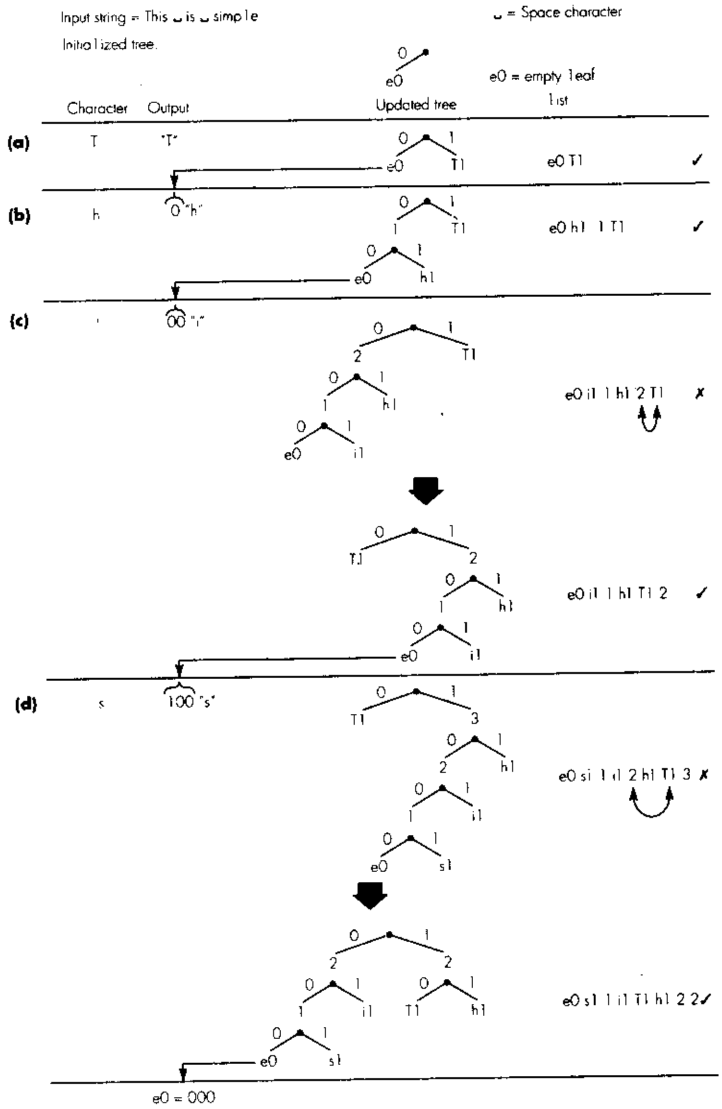


Figure 3.6 Dynamic Huffman encoding algorithm.

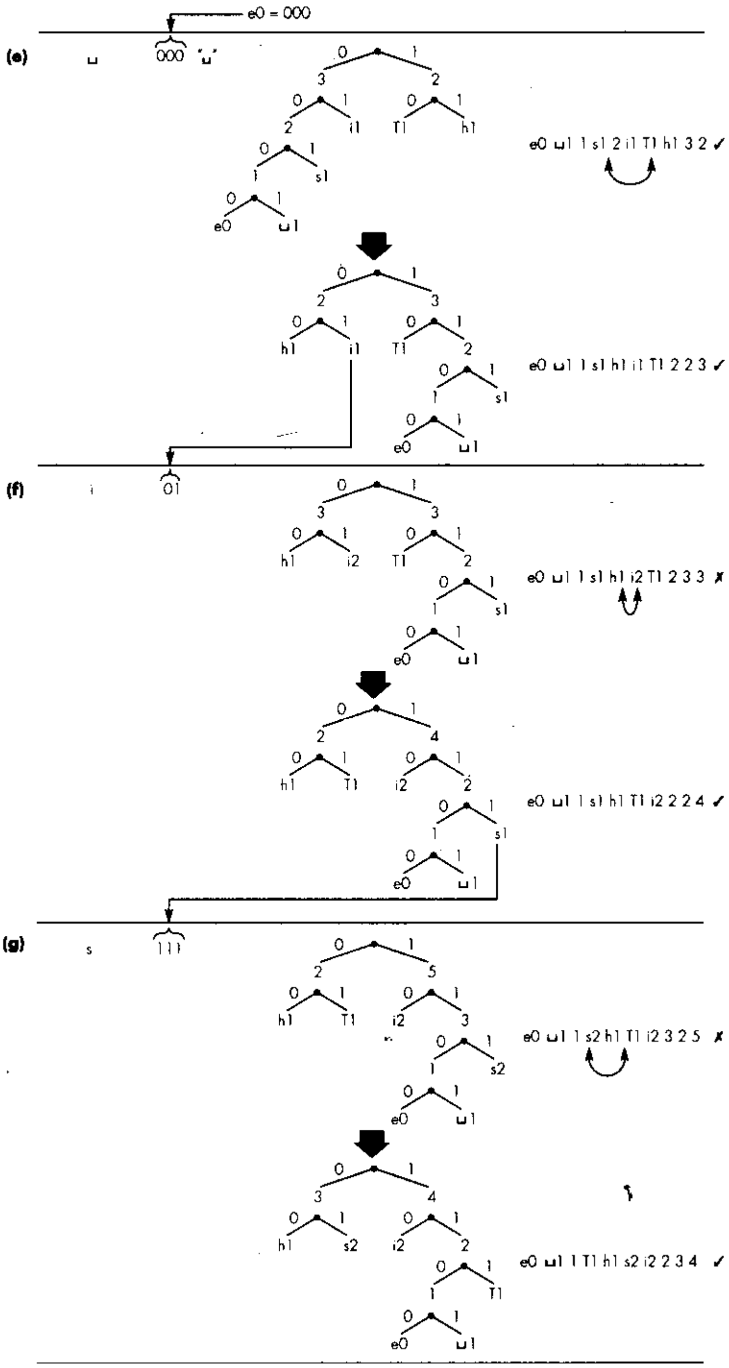

e0 = 000
(e)
␣
000 '␣'
e0 ␣1 1 s1 2 i1 T1 h1 3 2 ✓
e0 ␣1 1 s1 h1 i1 T1 2 2 3 ✓
(f)
i
01
e0 ␣1 1 s1 h1 i2 T1 2 3 3 ✗
e0 ␣1 1 s1 h1 T1 i2 2 2 4 ✓
(g)
s
111
e0 ␣1 1 s2 h1 T1 i2 3 2 5 ✗
e0 ␣1 1 T1 h1 s2 i2 2 3 4 ✓

Figure 3.6 Continued

The encoder then starts by reading the first character **T** and, since the tree is empty, it sends this in its uncompressed – say, ASCII – form. This is shown as "T" in the figure. The character is then assigned to the 1-branch of the root and, since this is the first occurrence of this character, it is shown as T1 in the tree. On reception, since the decoder's tree is also empty, it interprets the received bit string as an uncompressed character and proceeds to assign the character to its tree in the same way (Figure 3.6(a)).

For each subsequent character, the encoder first checks whether the character is already present in the tree. If it is, then the encoder sends the current codeword for the character in the normal way, the codeword being determined by the position of the character in the tree. If it is not present, then the encoder sends the current codeword for the empty leaf – again determined by its position in the tree – followed by the uncompressed codeword for the character. Since the decoder has the same tree as the encoder, it can readily deduce from the received bit string whether it is the current codeword of a (compressed) character or that of the empty leaf followed by the character in its uncompressed form.

The encoder and decoder proceed to update their copy of the tree based on the last character that has been transmitted/received. If it is a new character, the existing empty leaf node in the tree is replaced with a new branch node, the empty leaf being assigned to the 0-branch and the character to the 1-branch (Figure 3.6(b)).

If the character is already present in the tree, then the frequency of occurrence of the leaf node is incremented by unity. On doing this, the position of the leaf node may not now be in the optimum position in the tree. Hence each time the tree is updated – either by adding a new character or by incrementing the frequency of occurrence of an existing character – both the encoder and decoder check, and if necessary modify, the current position of all the characters in the tree.

To ensure that both the encoder and decoder do this in a consistent way, they first list the weights of the leaf and branch nodes in the updated tree from left to right and from bottom to top starting at the empty leaf. If they are all in weight order, all is well and the tree is left unchanged. If there is a node out of order, the structure of the tree is modified by exchanging the position of this node with the other node in the tree – together with its branch and leaf nodes – to produce an incremented weight order. The first occurrence is in Figure 3.6(c) and other examples are in parts (d)–(g).

The steps followed when a character to be transmitted has previously been sent are shown in Figure 3.6(f). At this point, the character to be transmitted is **i** and when the encoder searches the tree, it determines that **i** is already present and transmits its existing codeword – 01. The encoder then increments the character's weight – frequency of occurrence – by unity to i2 and updates the position of the modified node as before. Another example is shown in Figure 3.6(g) when the character **s** is to be transmitted.

We can deduce from this example that the savings in transmission bandwidth start only when characters begin to repeat themselves. In practice, the savings with text files can be significant, and dynamic Huffman coding is now used in a number of communication applications that involve the transmission of text.

3.3.3 Arithmetic coding

As we can deduce from Examples 3.1 and 3.2, Huffman coding achieves the Shannon value only if the character/symbol probabilities are all integer powers of ½. Clearly, in many instances, this is not the case and hence the set of codewords produced are rarely optimum. In contrast, the codewords produced using arithmetic coding always achieve the Shannon value. Arithmetic coding, however, is more complicated than Huffman coding and so we shall limit our discussion of it to the basic static coding mode of operation.

To illustrate how the coding operation takes place, consider the transmission of a message comprising a string of characters with probabilities of:

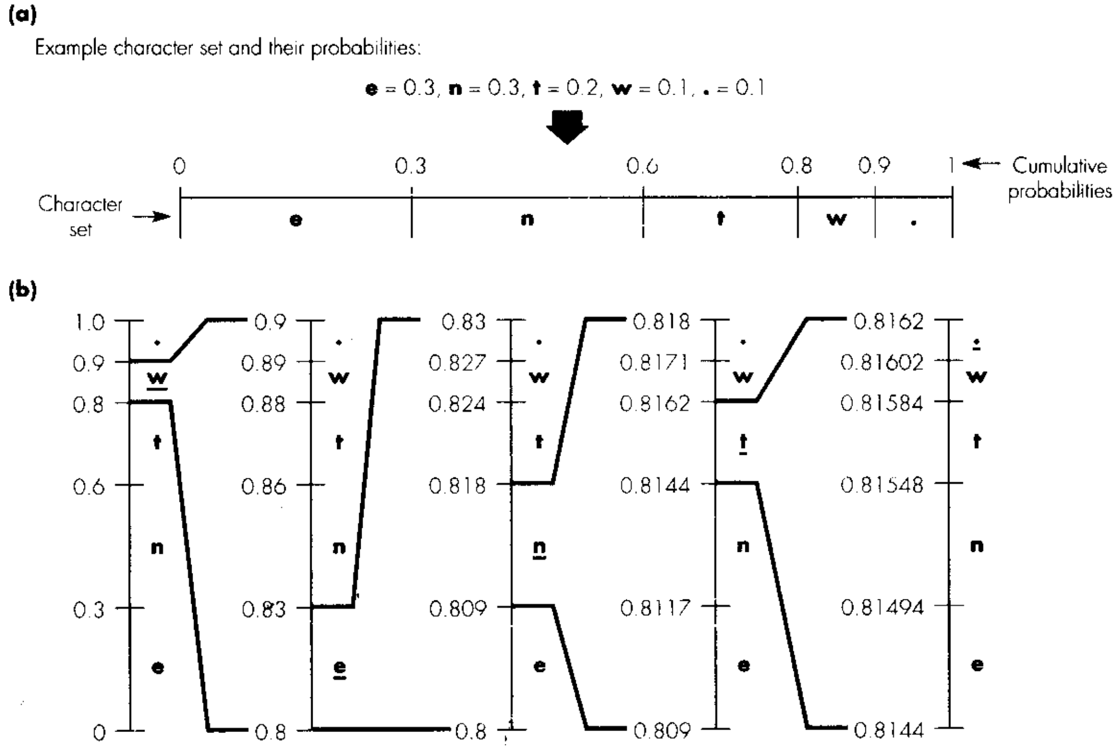
e = 0.3, **n** = 0.3, **t** = 0.2, **w** = 0.1, **.** = 0.1

At the end of each character string making up a message, a known character is sent which, in this example, is a period **.**. When this is decoded at the receiving side, the decoder interprets this as the end of the string/message.

Unlike Huffman coding which was a separate codeword for each character, arithmetic coding yields a single codeword for each encoded string of characters. The first step is to divide the numeric range from 0 to 1 into a number of different characters present in the message to be sent – including the termination character – and the size of each segment by the probability of the related character. Hence the assignments for our set of five characters may be as shown in Figure 3.7(a).

As we can see, since there are only five different characters, there are five segments, the width of each segment being determined by the probability of the related character. For example, the character **e** has a probability of 0.3 and hence is assigned the range from 0.0 to 0.3, the character **n** – which also has a probability of 0.3 – the range from 0.3 to 0.6, and so on. Note, however, that an assignment in the range, say, 0.8 to 0.9, means that the probability in the cumulative range is from 0.8 to 0.8999... . Once this has been done, we are ready to start the encoding process. An example is shown in Figure 3.7(b) and, in this example, we assume the character string/message to be encoded is the single word **went.**.

The first character to be encoded **w** is in the range 0.8 to 0.9. Hence, as we shall see, the final (numeric) codeword is a number in the range 0.8 to 0.8999 ... since each subsequent character in the string subdivides the range 0.8 to 0.9 into progressively smaller segments each determined by the probabilities of the characters in the string.



Encoded version of the character string **went.** is a single codeword in the range 0.816 02 ≤ codeword < 0.8162

Figure 3.7 Arithmetic coding principles: (a) example character set and their range assignments; (b) encoding of the string went..

As we can see in the example, since **w** is the first character in the string, the range 0.8 to 0.9 is itself subdivided into five further segments, the width of each segment again determined by the probabilities of the five characters. Hence the segment for the character **e**, for example, is from 0.8 to 0.83 (0.8 + 0.3 × 0.1), the character **n** from 0.83 to 0.86 (0.83 + 0.3 × 0.1), and so on.

The next character in the string is **e** and hence its range (0.8 to 0.83) is again subdivided into five segments. With the new assignments, therefore, the character **e** has a range from 0.8 to 0.809 (0.8 + 0.3 × 0.03), the character **n** from 0.809 to 0.818 (0.809 + 0.3 × 0.03), and so on. This procedure continues until the termination character **.** is encoded. At this point, the segment range of **.** is from 0.816 02 to 0.8162 and hence the codeword for the complete string is any number within the range:

$$0.81602 \leq \text{codeword} > 0.8162$$

Static mode

rx knows → set of char present in encoded msg

→ segment to which each char [illegible]

decoder follow same procedure as encoder to determine the char strings relating to each rx codeword

eg

In the static mode, the decoder knows the set of characters that are present in the encoded messages it receives as well as the segment to which each character has been assigned and its related range. Hence with this as a start point, the decoder can follow the same procedure as that followed by the encoder to determine the character string relating to each received codeword. For example, if the received codeword is, say, 0.8161, then the decoder can readily determine from this that the first character is w since it is the only character within the range 0.8 to 0.9. It then expands this interval as before and determines that the second character must be e since 0.8161 is within the range 0.8 to 0.83. This procedure then repeats until it decodes the known termination character . at which point it has recreated the, say, ASCII string relating to **went.** and passes this on for processing.

As we can deduce from this simple example, the number of decimal digits in the final codeword increases linearly with the number of characters in the string to be encoded. Hence the maximum number of characters in a string is determined by the precision with which floating-point numbers are represented in the source and destination computers. As a result, a complete message may be first fragmented into multiple smaller strings. Each string is then encoded separately and the resulting set of codewords sent as a block of (binary) floating-point numbers each in a known format. Alternatively, **binary arithmetic coding** can be used but, as we indicated earlier, this is outside the scope of the book. Further details relating to arithmetic coding can be found in the bibliography for this chapter at the end of the book.

3.3.4 Lempel–Ziv coding

instead of single char for coding op → uses string of char

a table containing all possible char strings words that occur in the text to be tx is held by ~~deco~~ & encoder

↓

as words occur instead of sending the words as set of indi. ASCII codewords encoder → sends only the index of where its stored in table

decoder – access the table

↓

reconstruct orig. t/g

↓

table used as dictionary

LZ algo → dictionary based compression algorithm

The Lempel–Ziv (LZ) compression algorithm, instead of using single characters as the basis of the coding operation, uses strings of characters. For example, for the compression of text, a table containing all the possible character strings – for example words – that occur in the text to be transferred is held by both the encoder and decoder. As each word occurs in the text, instead of sending the word as a set of individual – say, ASCII – codewords, the encoder sends only the index of where the word is stored in the table and, on receipt of each index, the decoder uses this to access the corresponding word/string of characters from the table and proceeds to reconstruct the text into its original form. Thus the table is used as a dictionary and the LZ algorithm is known as a **dictionary-based** compression algorithm.

Most word-processing packages have a dictionary associated with them which is used for both spell checking and for the compression of text. Typically, they contain in the region of 25 000 words and hence 15 bits – which has 32 768 combinations – are required to encode the index. To send the word "multimedia" with such a dictionary would require just 15 bits instead of 70 bits with 7-bit ASCII codewords. This results in a compression ratio of 4.7:1. Clearly, shorter words will have a lower compression ratio and longer words a higher ratio.

Typically – 25000 words ↓ 15 bits (32768) reqd to encode

multimedia → 15 bits (dictionary) – 70 bits with 7 bits ASCII codewords compression ratio 4.7:1

Example 3.3

[illegible] to be compressed is available prior to its trans- [illegible] number of characters per word is 6, and the [illegible] 4096 words, derive the average compression [illegible] to using 7-bit ASCII codewords.

[illegible]

[illegible] dictionary with an index of n bits can contain up to 2^n words. Now $4096 = 2^{12}$ and hence an index of 12 bits is required.

Using 7-bit ASCII codewords and an average of 6 characters per word requires 42 bits.

Hence compression ratio = 42/12 = 3.5:1.

As with the other static coding methods, the basic requirement with the LZ algorithm is that a copy of the dictionary is held by both the encoder and the decoder. Although this is acceptable for the transmission of text which has been created using a standard word-processing package, it can be relatively inefficient if the text to be transmitted comprises only a small subset of the words stored in the dictionary. Hence a variation of the LZ algorithm has been developed which allows the dictionary to be built up dynamically by the encoder and decoder as the compressed text is being transferred. In this way, the size of the dictionary is often a better match to the number of different words in the text being transmitted than if a standard dictionary was used.

variation of LZ algo
dict → built up dynamically
as text is being transferred
size → reduced

3.3.5 Lempel–Ziv–Welsh coding

The principle of the Lempel–Ziv–Welsh (LZW) coding algorithm is for the encoder and decoder to build the contents of the dictionary dynamically as the text is being transferred. Initially, the dictionary held by both the encoder and decoder contains only the character set – for example ASCII – that has been used to create the text. The remaining entries in the dictionary are then built up dynamically by both the encoder and decoder and contain the words that occur in the text. For example, if the character set comprises 128 characters and the dictionary is limited to, say, 4096 entries, then the first 128 entries would contain the single characters that make up the character set and the remaining 3968 entries would each contain strings of two or more characters that make up the words in the text being transferred. As we can see, the more frequently the words stored in the dictionary occur in the text, the higher the level of compression.

In order to describe how the dictionary is built up, let us assume that the text to be compressed starts with the string:

initially contains only
char set
that's been used to
create the text.
remaining entries
built up.
eg: If char set
comprises 128 char
limited
and dictionary
to 4096 entries
f. 128. single char
rest → words.
the more freq the word
occurs → more the
compression.

This is simple as it is ...

Since the idea is for the dictionary to contain only words, then only strings of characters that consist of alphanumeric characters are stored in the dictionary and all the other characters in the set are interpreted as word delimiters.

Initially, the dictionary held by both the encoder and decoder contains only the individual characters from the character set being used; for example, the 128 characters in the ASCII character set. Hence the first word in the example text is sent by the encoder using the index of each of the four characters *T*, *h*, *i*, and *s*. At this point, when the encoder reads the next character from the string – the first space (SP) character – it determines that this is not an alphanumeric character. It therefore transmits the character using its index as before but, in addition, interprets it as terminating the first word and hence stores the preceding four characters in the next available (free) location in the dictionary. Similarly the decoder, on receipt of the first five indices/codewords, reads the character stored at each index and commences to reconstruct the text. When it determines that the fifth character is a space character, it interprets this as a word delimiter and proceeds to store the word *This* in its dictionary.

A first char.
↓
detects words
stores it
([illegible])

decoder reads the char stored at each index & commences to reconstruct the text.

The same procedure is followed by both the encoder and decoder for transferring the other words except the encoder, prior to sending each word in the form of single characters, first checks to determine if the word is currently stored in its dictionary and, if it is, it sends only the index for the word. Similarly the decoder, since it also has the word stored in its dictionary, uses the index to access the string of characters that make up the word. So with the example text string, after the space character following the second occurrence of the word *is*, the contents of the dictionary held by both the encoder and the decoder will be as shown in Figure 3.8(a). As we can see, since this is the second occurrence of the word *is*, it is transferred using only the index of where it is stored in the dictionary (129).

encoder prior to [illegible] checks if the word [illegible] in the dictionary

As we can deduce from this example, a key issue in determining the level of compression that is achieved, is the number of entries in the dictionary since this, in turn, determines the number of bits that are required for the index. With a static dictionary, the number of entries is fixed and, for the example we identified earlier, a dictionary containing 25 000 words requires 15 bits to encode the index. When building the dictionary dynamically, however, the question arises as to how many entries should be provided for the dictionary. Clearly, if too few entries are provided then the dictionary will contain only a subset of the words that occur in the text while if too many are provided, then it will contain empty spaces which, in turn, makes the index unnecessarily long. In order to optimize the number of bits used for the index, at the commencement of each transfer the number of entries is set to a relatively low value but, should the available space become full, then the number of entries is allowed to increase incrementally.

> when building dict. dynamically Q is how many entries to be provided?

If few then empty spaces
↓
makes index long unnecessarily
so its set to low value
& if full its increased incrementally

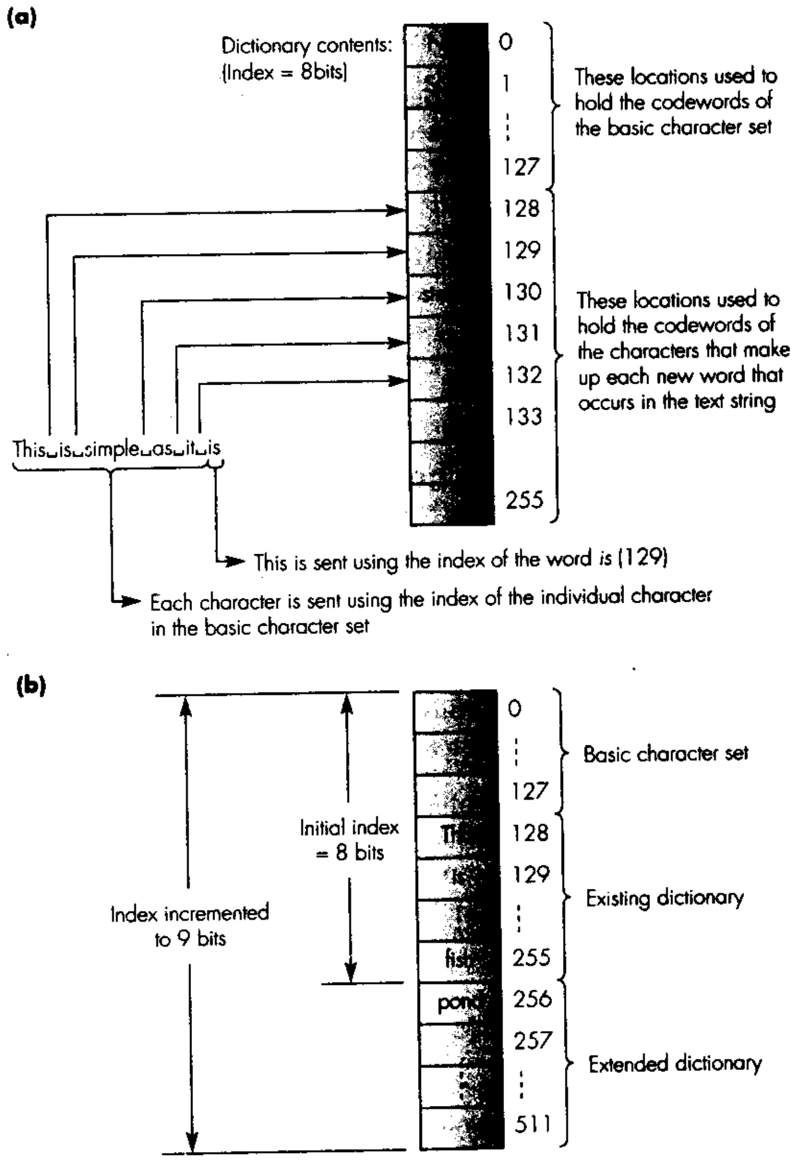

(a)
Dictionary contents:
(Index = 8bits)
0
1
127
128
129
130
131
132
133
255
These locations used to hold the codewords of the basic character set
These locations used to hold the codewords of the characters that make up each new word that occurs in the text string
This␣is␣simple␣as␣it␣is
This is sent using the index of the word is (129)
Each character is sent using the index of the individual character in the basic character set
(b)
Initial index = 8 bits
Index incremented to 9 bits
0
127
128
129
255
256
257
511
Basic character set
Existing dictionary
Extended dictionary

Figure 3.8 LZW compression algorithm: (a) basic operation; (b) dynamically extending the number of entries in the dictionary.

For example, in an application that uses 128 characters in the basic character set, then both the encoder and decoder would start with, say, 256 entries in the dictionary. This requires an index/codeword length of 8 bits and the dictionary would provide space for the 128 characters in the character set and a further 128 locations for words that occur in the text. Should this number of locations become insufficient, on detecting this, both the encoder and decoder would double the size of their dictionary to 512 locations. Clearly, this necessitates an index length of 9 bits and so from this point, the encoder uses 9-bit codewords. However, since the decoder has also doubled the size of its own directory, it expects 9-bit codewords from this point. In this way, the number of entries in the dictionary more accurately reflects the number of different words in the text being transferred and hence optimizes the number of bits used for each index/codeword.

The procedure is shown in diagrammatic form in Figure 3.8(b). In this example it is assumed that the last entry in the existing table at location 255 is the word *fish* and the next word in the text that is not currently in the dictionary is *pond.*

3.4 Image compression

Recall from Section 2.4 how images can be of two basic types: computer-generated (also known as graphical) images and digitized images (of both documents and pictures). Although both types are displayed (and printed) in the form of a two-dimensional matrix of individual picture elements, normally a graphical image is represented differently in the computer file system. Typically, this is in the form of a program (written in a particular graphics programming language) and, since this type of representation requires considerably less memory (and hence transmission bandwidth) than the corresponding matrix of picture elements, whenever possible, graphics are transferred across a network in this form. In the case of digitized documents and pictures, however, once digitized, the only form of representation is as a two-dimensional matrix of picture elements.

In terms of compression, when transferring graphical images which are represented in their program form, a lossless compression algorithm must be used similar, for example, to those in the last section. However, when the created image/graphic is to be transferred across the network in its bit-map form, then this is normally compressed prior to its transfer. There are a number of different compression algorithms and associated file formats in use and we shall describe two of these in the next two sections.

To transfer digitized images a different type of compression algorithm must normally be employed and, in practice, two different schemes are used. The first is based on a combination of run-length and statistical encoding. Hence it is lossless and is used for the transfer of the digitized documents generated by scanners such as those used in facsimile machines. The second is based on a combination of transform, differential, and run-length encoding and has been developed for the compression of both bitonal and color digitized pictures. Since there is an international standard associated with both schemes, we shall limit our discussion to these two schemes.

images

comp generated (graphical) | digitized (doc & pict)

displayed in form of a 2D matrix of pixels

program | vs 2D Matrix

less memory & tx BW

∴ pgm is sent across n/w

only 2D matrix form

graphical → lossless compression algorithm must be used.

digitized → diff type of compression

comb of Run length & statistical lossless → eg: FAX

comb. of tx, diff & run length coding eg: comp of bitonal & color digitized picture

3.4.1 Graphics interchange format

The **graphics interchange format** (GIF) is used extensively with the Internet for the representation and compression of graphical images. Although color images comprising 24-bit pixels are supported – 8 bits each for R, G, and B – GIF reduces the number of possible colors that are present by choosing the 256 colors from the original set of 2^{24} colors that match most closely those used in the original image. The resulting table of colors therefore consists of 256 entries, each of which contains a 24-bit color value. Hence instead of sending each pixel as a 24-bit value, only the 8-bit index to the table entry that contains the closest match color to the original is sent. This results in a compression ratio of 3:1. The table of colors can relate either to the whole image – in which case it is referred to as the **global color table** – or to a portion of the image, when it is referred to as a **local color table**. The contents of the table are sent across the network – together with the compressed image data and other information such as the screen size and aspect ratio – in a standardized format. The principles of the scheme are shown in Figure 3.9(a).

As we show in Figure 3.9(b), the LZW coding algorithm can be used to obtain further levels of compression. We described this earlier in Section 3.3.5 when we discussed text compression and, in the case of image compression, this works by extending the basic color table dynamically as the compressed image data is being encoded and decoded. As with text compression, the occurrence of common strings of pixel values – such as long strings of the same color – are detected and these are entered into the color table after the 256 selected colors. However in this application, since each entry in the color table comprises 24 bits, in order to save memory, to represent each string of pixel values just the corresponding string of 8-bit indices to the basic color table are used. If we limit each entry in the table to 24 bits, then this will allow common strings comprising three pixel values to be stored in each location of the extended table. Normally, since the basic table contains 256 entries, an initial table size of 512 entries is selected which allows for up to 256 common strings to be stored. As with text compression, however, should more strings be found, then the number of entries in the table is allowed to increase incrementally by extending the length of the index by 1 bit.

GIF also allows an image to be stored and subsequently transferred over the network in an interlaced mode. This can be useful when transferring images over either low bit rate channels or the Internet which provides a variable transmission rate. With this mode, the compressed image data is organized so that the decompressed image is built up in a progressive way as the data arrives. To achieve this, the compressed data is divided into four groups as shown in Figure 3.10 and, as we can see, the first contains 1/8 of the total compressed image data, the second a further 1/8, the third a further 1/4, and the last the remaining 1/2.

(1)
(2)
color images
comprising 24bit
pixels are supported
8-R/G/B
↓
no. of possible colors
2^{24}
GIF uses only 256
that match most closely
those in the original
image
↓
table → 256 entries each
rep. a 24bit value
only the 8bit index
the table entry
containing the closest
match is sent.
↓
compression ratio 3:1

if table relates to the
whole image → global
color
table
portion → local

contents of the table
+
compressed image data
+
info → screen size
aspect ratio.
are sent across the
network.

LZW coding → further compression

GIF image store & transferred over
the n/w in an interlaced
mode
↓
useful when to images over [illegible]
low bit rate [illegible] int
which provide variable data rate.

comp image data is
organized so that
the decompressed image
is progressively built
[illegible] data

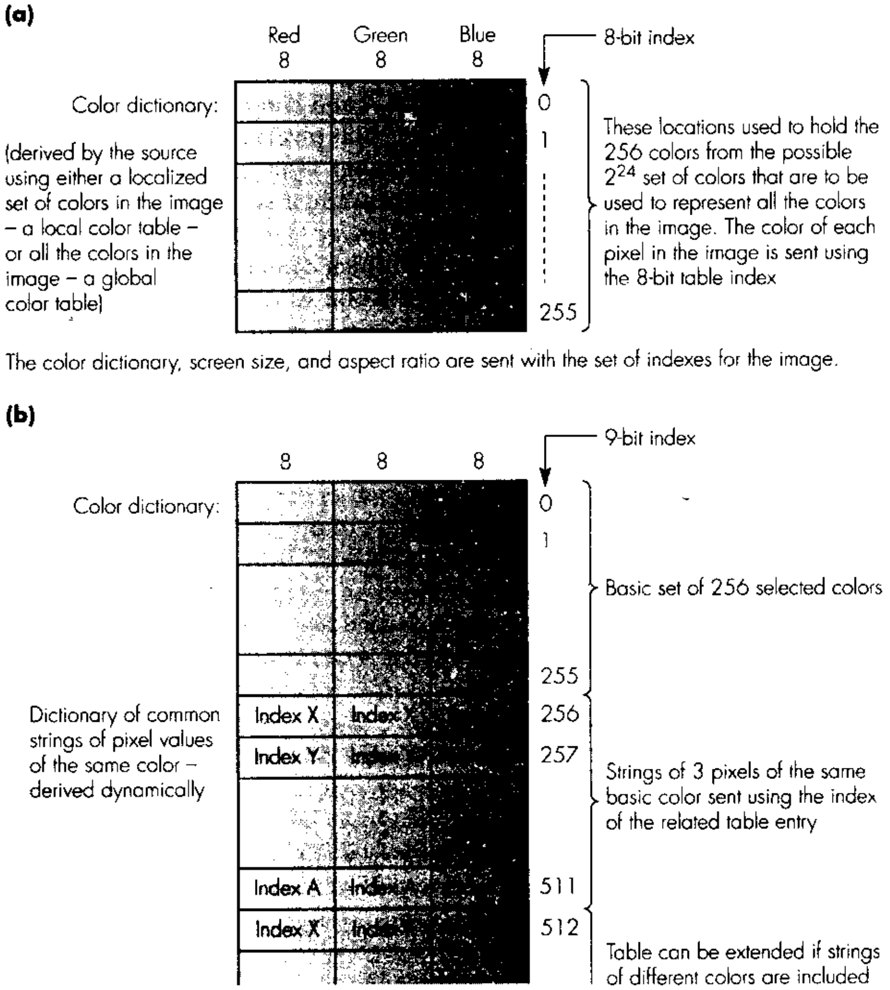

(a)
Red 8
Green 8
Blue 8
8-bit index
Color dictionary:
(derived by the source using either a localized set of colors in the image – a local color table – or all the colors in the image – a global color table)
0
1
255
These locations used to hold the 256 colors from the possible 2^24 set of colors that are to be used to represent all the colors in the image. The color of each pixel in the image is sent using the 8-bit table index
The color dictionary, screen size, and aspect ratio are sent with the set of indexes for the image.
(b)
8
8
8
9-bit index
Color dictionary:
0
1
255
Basic set of 256 selected colors
Dictionary of common strings of pixel values of the same color – derived dynamically
Index X
Index Y
Index A
Index X
Index
256
257
511
512
Strings of 3 pixels of the same basic color sent using the index of the related table entry
Table can be extended if strings of different colors are included

Figure 3.9 GIF compression principles: (a) basic operational mode; (b) dynamic mode using LZW coding.

3.4.2 Tagged image file format

(1)

The **tagged image file format** (**TIFF**) is also used extensively. It supports pixel resolutions of up to 48 bits – 16 bits each for R, G, and B – and is intended for the transfer of both images and digitized documents. The image data, therefore, can be stored – and hence transferred over the network – in a number of different formats. The particular format being used is indicated by a code number and these range from the uncompressed format (code number 1) through to LZW-compressed which is code number 5. Code numbers 2, 3, and 4 are intended for use with digitized documents. These use the same

- supports pixel resol upto 48 bits → 16 each
- tx of image & digitized doc
- image data can be stored & hence tx to the n/w in no of formats
- format used → indicated by a code no
uncompressed (1) | 2, 3, 4 → digitized doc
LZ comp (5) | same compression as FAX machine

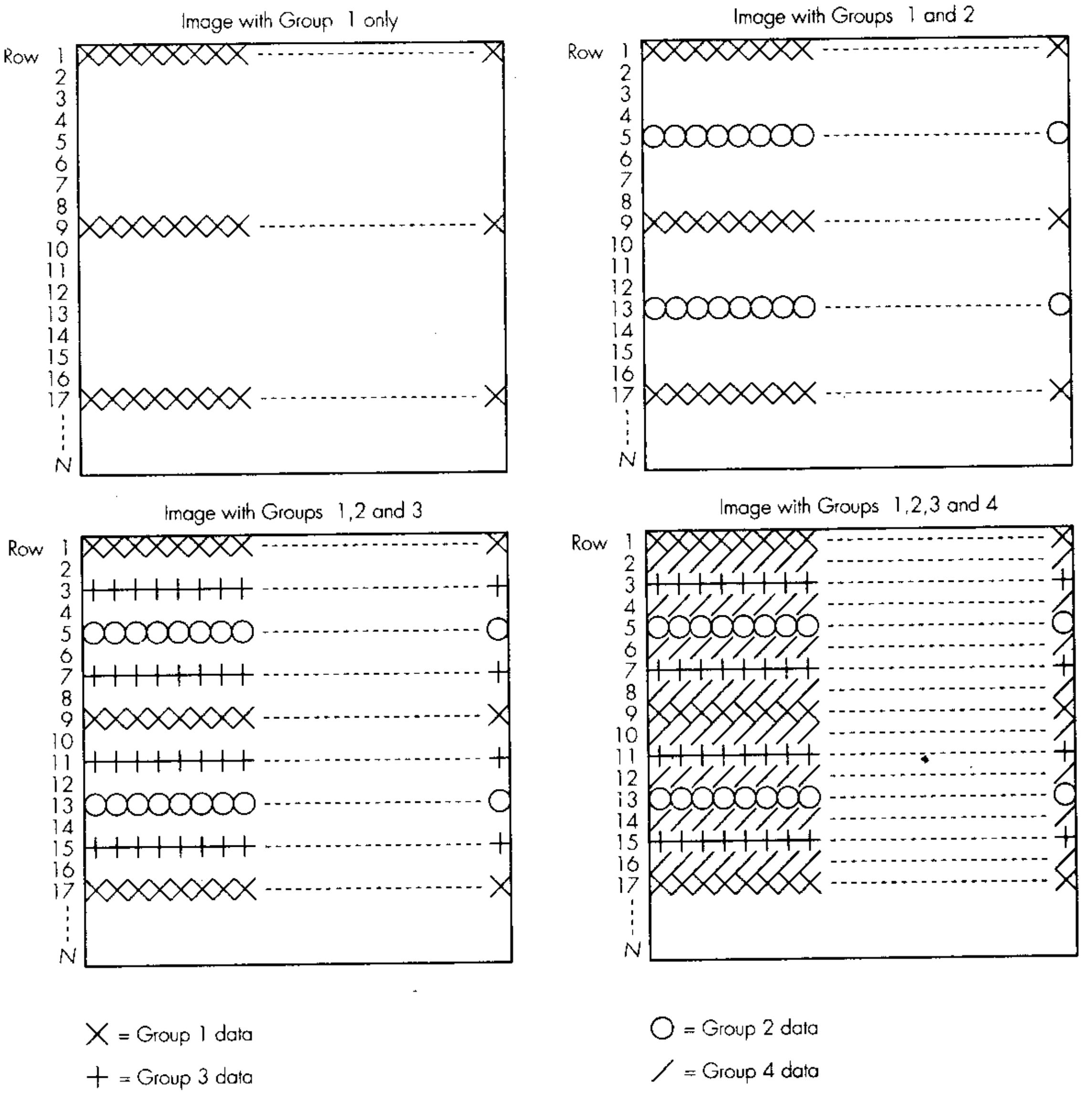


Figure 3.10 GIF interlaced mode.

compression algorithms that are used in facsimile machines which we discuss in the next section.

The LZW compression algorithm that is used is the same as that used with GIF. It starts with a basic color table containing 256 colors and the table can be extended to contain up to 4096 entries containing common strings of pixels in the image being transferred. Again, a standard format is used for the transfer of both the color table and the compressed image data.

3.4.3 Digitized documents

We described the principle of operation of the scanners used in facsimile machines to digitize bitonal images (such as a printed document) in Section 2.4.3. The digital representation of a scanned page was shown in Figure 2.11(b) and, even though only a single binary bit is used to represent each picture element, with the resolutions used, this produces an uncompressed bit stream of the order of 2 Mbits. In most cases this must be transferred using modems and the public switched telephone network. The relatively low bit rates available with modems means that it would be both costly and time consuming to transfer a total document comprising many pages in this basic form.

most doc
scanned lines → long strings of (white picture element)
while others → mix of black/pel[illegible]

FAX machines
used with public carrier
n/ws T2 (Group1), T3 (Group2) → earlier
T4 (Group3) → with modems to use over an analog PSTN
T6 (Group4) → all digital for use with digi n/w ISDN

① With most documents, many scanned lines consist only of long strings of white picture elements – pels – while others comprise a mix of long strings of white and long strings of black pels. Since facsimile machines are normally used with public carrier networks, the ITU-T has produced standards relating to them. These are T2 (Group 1), T3 (Group 2), T4 (Group 3), and T6 (Group 4). The first two are earlier standards and are now rarely used. The last two, however, both operate digitally; Group 3 with modems for use with an analog PSTN, and Group 4 all-digital for use with digital networks such as the ISDN. Both use data compression, and compression ratios in excess of 10:1 are common with most document pages. The time taken to transmit a page is reduced to less than a minute with Group 3 machines and, because of the added benefit of a higher transmission rate (64 kbps), to less than a few seconds with a Group 4 machine.

both use data comp
10:1 comp ratio
time to tx
less than a min | less than few secs due to high tx rate (64kbps)

As part of the standardization process, extensive analyses of typical scanned document pages were made. Tables of codewords were produced based on the relative frequency of occurrence of the number of contiguous white and black pels found in a scanned line. The resulting codewords are fixed and grouped into two separate tables: the **termination-codes table** and the **make-up codes table**. The codewords in each table are shown in Figure 3.11.

Codewords in the termination-codes table are for white or black run-lengths of from 0 to 63 pels in steps of 1 pel; the make-up codes table contains codewords for white or black run-lengths that are multiples of 64 pels. A technique known as **overscanning** is used which means that all lines start with a minimum of one white pel. In this way, the receiver knows the first codeword always relates to white pels and then alternates between black and white. Since the scheme uses two sets of codewords (termination and make-up) they are known as **modified Huffman codes**. As an example, a run-length of 12 white pels is coded directly as 001000. Similarly, a run-length of 12 black pels is coded directly as 0000111. A run-length of 140 black pels, however, is encoded as 000011001000 + 0000111; that is, 128 + 12 pels. Run-lengths exceeding 2560 pels are encoded using more than one make-up code plus one termination code.

analysis of scanned pages
→ tables of codeword (relative freq. of occurence of no. of black/white pels in a scanned line)
→ 2 seperate tables → termination codes table
→ make up codes table

There is no error-correction protocol with Group 3. From the list of codewords, we can deduce that if one or more bits is corrupted during its

overscanning all the lines start with one white pel
hence rx knows ...
modified [illegible] of codewords)

(a)

White run-length	Code-word	Black run-length	Code-word
0	00110101	0	0000110111
1	000111	1	010
2	0111	2	11
3	1000	3	10
4	1011	4	011
5	1100	5	0011
6	1110	6	0010
7	1111	7	00011
8	10011	8	000101
9	10100	9	000100
10	00111	10	0000100
11	01000	11	0000101
12	001000	12	0000111
13	000011	13	00000100
14	110100	14	00000111
15	110101	15	000011000
16	101010	16	0000010111
17	101011	17	0000011000
18	0100111	18	0000001000
19	0001100	19	00001100111
20	0001000	20	00001101000
21	0010111	21	00001101100
22	0000011	22	00000110111
23	0000100	23	00000101000
24	0101000	24	00000010111
25	0101011	25	00000011000
26	0010011	26	000011001010
27	0100100	27	000011001011
28	0011000	28	000011001100
29	00000010	29	000011001101
30	00000011	30	000001101000
31	00011010	31	000001101001
32	00011011	32	000001101010
33	0010010	33	000001101011
34	00010011	34	000011010010
35	00010100	35	000011010011
36	00010101	36	000011010100
37	00010110	37	000011010101
38	00010111	38	000011010110
39	00101000	39	000011010111
40	00101001	40	000001101100
41	00101011	41	000001101101
42	00101011	42	000011011010
43	00101100	43	000011011011
44	00101101	44	000001010100
45	00000100	45	000001010101
46	00000101	46	000001010110
47	00001010	47	000001010111
48	00001011	48	000001100100
49	01010010	49	000001100101
50	01010011	50	000001010010
51	01010100	51	000001010011
52	01010101	52	000000100100
53	00100100	53	000000110111
54	00100101	54	000000111000
55	01011000	55	000000100111

(a) cont.

White run-length	Code-word	Black run-length	Code-word
56	01011001	56	000000101000
57	01011010	57	000001011000
58	01011011	58	000001011001
59	01001010	59	000000101011
60	01001011	60	000000101100
61	00110010	61	000001011010
62	00110011	62	000001100110
63	00110100	63	000001100111

(b)

White run-length	Code-word	Black run-length	Code-word
64	11011	64	0000001111
128	10010	128	000011001000
192	010111	192	000011001001
256	0110111	256	000001011011
320	00110110	320	000000110011
384	00110111	384	000000110100
448	01100100	448	000000110101
512	01100101	512	0000001101100
576	01101000	576	0000001101101
640	01100111	640	0000001001010
704	011001100	704	0000001001011
768	011001101	768	0000001001100
832	011010010	832	0000001001101
896	011010011	896	0000001110010
960	011010100	960	0000001110011
1024	011010101	1024	0000001110100
1088	011010110	1088	0000001110101
1152	011010111	1152	0000001110110
1216	011011000	1216	0000001110111
1280	011011001	1280	0000001010010
1344	011011010	1344	0000001010011
1408	011011011	1408	0000001010100
1472	010011000	1472	0000001010101
1536	010011001	1536	0000001011010
1600	010011010	1600	0000001011011
1664	011000	1664	0000001100100
1728	010011011	1728	0000001100101
1792	00000001000	1792	00000001000
1856	00000001100	1856	00000001100
1920	00000001101	1920	00000001101
1984	000000010010	1984	000000010010
2048	000000010011	2048	000000010011
2112	000000010100	2112	000000010100
2176	000000010101	2176	000000010101
2240	000000010110	2240	000000010110
2304	000000010111	2304	000000010111
2368	000000011100	2368	000000011100
2432	000000011101	2432	000000011101
2496	000000011110	2496	000000011110
2560	000000011111	2560	000000011111
EOL	000000000001	EOL	000000000001

Figure 3.11 ITU–T Group 3 and 4 facsimile conversion codes: (a) termination-codes, (b) make-up codes.

transmission through the network, the receiver will start to interpret subsequent codewords on the wrong bit boundaries. The receiver thus becomes unsynchronized and cannot decode the received bit string. To enable the receiver to regain synchronism, each scanned line is terminated with a known **end-of-line (EOL) code**. In this way, if the receiver fails to decode a valid codeword after the maximum number of bits in a codeword have been scanned (parsed), it starts to search for the EOL pattern. If it fails to decode an EOL after a preset number of lines, it aborts the reception process and informs the sending machine. A single EOL precedes the codewords for each scanned page and a string of six consecutive EOLs indicates the end of each page.

If [illegible] bit corrupted during tx (rx → interprets subsequent code on wrong boundaries unsync & can't decode the bit string)

To regain sync EOL code is used. each line is terminated with it.

If rx fails to decode valid code for the bits it searches for the EOL pattern

if EOL not there stops reception informs sender

Because each scanned line is encoded independently, the T4 coding scheme is known as a **one-dimensional coding** scheme. As we can conclude, it works satisfactorily providing the scanned image contains significant areas of white or black pels which occur, for example, where documents consist of letters and line drawings. Documents containing photographic images, however, are not satisfactory as the different shades of black and white are represented by varying densities of black and white pels. This, in turn, results in a large number of very short black or white run-lengths which, with the T4 coding scheme, can lead to a **negative compression ratio**; that is, more bits are needed to send the scanned document in its compressed form than are needed in its uncompressed form.

For this reason the alternative T6 coding scheme has been defined. It is an optional feature in Group 3 facsimile machines but is compulsory in Group 4 machines. When supported in Group 3 machines, the EOL code at the end of each (compressed) line has an additional tag bit added. If this is a binary 1 then the next line has been encoded using the T4 coding scheme, if it is a 0 then the T6 coding scheme has been used. The latter is known as **modified-modified READ (MMR) coding**. It is also known as **two-dimensional** or **2D coding** since it identifies black and white run-lengths by comparing adjacent scan lines. READ stands for **relative element address designate**, and it is "modified" since it is a modified version of an earlier (modified) coding scheme.

in Group 3 EOL code at the end of each compressed line

↓

additional tag bit

1 — Next line encoded using T4

0 — T6 encoding scheme — MMR coding (OR) 2D coding identifies ~~[illegible]~~ B/W run lengths by comparing adjacent scan lines

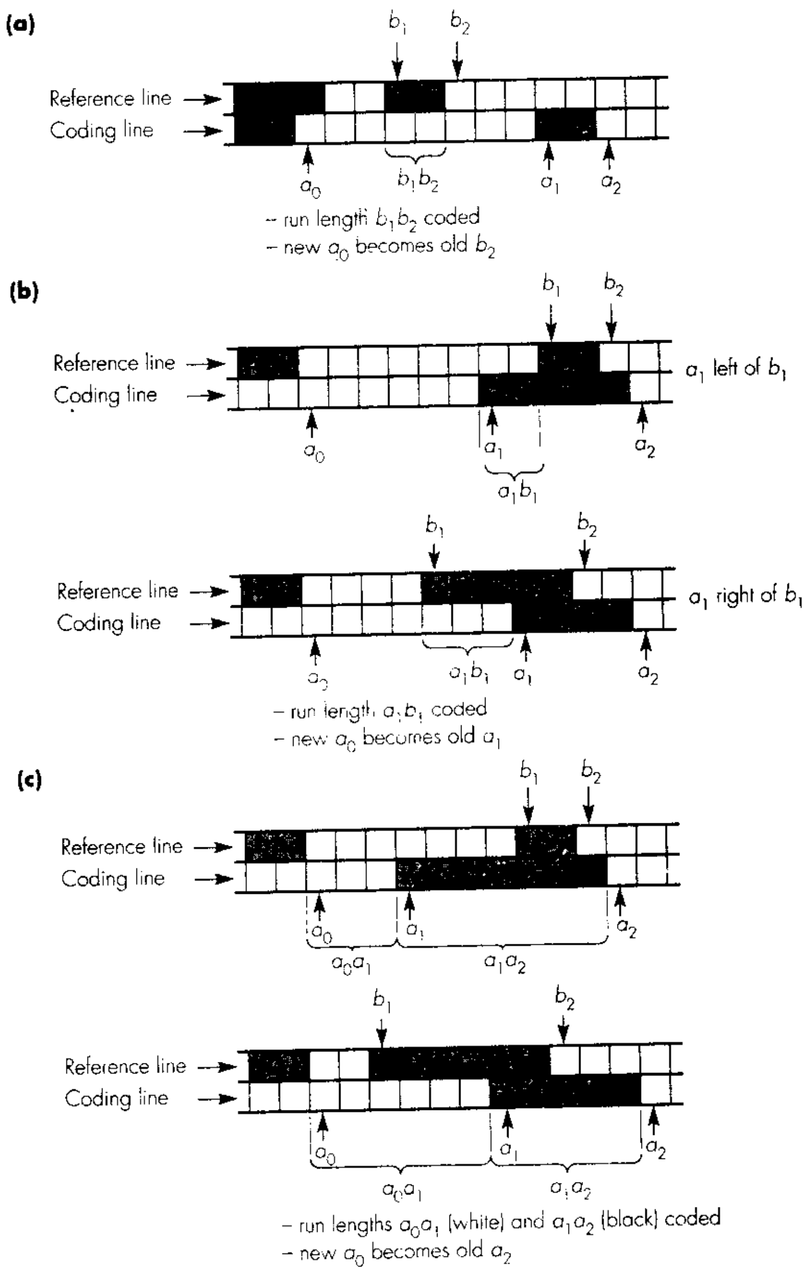
~~MMR coding~~

MMR coding exploits the fact that most scanned lines differ from the previous line by only a few pels. For example, if a line contains a black-run then the next line will normally contain the same run plus or minus up to three pels. With MMR coding the run-lengths associated with a line are identified by comparing the line contents, known as the **coding line** (**CL**), relative to the immediately preceding line, known as the **reference line** (**RL**). We always assume the first reference line to be an (imaginary) all-white line and the first line proper is encoded relative to this. The encoded line then becomes the reference line for the following line, and so on. To ensure that the complete page is scanned, the scanner head always starts to the left of the page, so each line always starts with an imaginary white pel.

We identify the run lengths associated with a coding line as one of three possibilities or **modes** relative to the reference line. Examples of the three modes are shown in Figure 3.12. The three modes are identified by the position of the next run-length in the reference line ($b_1 b_2$) relative to the start

→ exploits the fact that most scanned lines differ from previous line by only a [illegible] eg.

run lengths associated with coding line can be identified as one of the 3 possibilities or modes relative to reference line.



Note: a_0 is the first pel of a new codeword and can be black or white
a_1 is the first pel to the right of a_0 with a different color
b_1 is the first pel on the reference line to the right of a_0 with a different color
b_2 is the first pel on the reference line to the right of b_1 with a different color

Figure 3.12 Some example run-length possibilities: (a) pass mode; (b) vertical mode; (c) horizontal mode.

and end of the next pair of run-lengths in the coding line (a_0a_1 and a_1a_2). Note that the same procedure is used to encode the runs of both black and white pels. The three possibilities are:

run length (ref) – b_1b_2 is to left of next runlength (cod.) a_1a_2 — ie b_2 to left of a_1
if next pel on (coding) a_1 is dir. below b_2 → X pass mode

1 **Pass mode:** This is the case when the run-length in the reference line (b_1b_2) is to the left of the next run-length in the coding line (a_1a_2), that is, b_2 is to the left of a_1. An example is given in Figure 3.12(a) and, for this mode, the run-length b_1b_2 is coded using the codewords given in Figure 3.11. Note that if the next pel on the coding line, a_1, is directly below b_2 then this is not pass mode.

when RL in ref b_1b_2 overlaps the Next RL in coding line a_1a_2 by a max of ± 3 pels
diff runlength a_1b_1 is coded

2 **Vertical mode:** This is the case when the run-length in the reference line (b_1b_2) overlaps the next run-length in the coding line (a_1a_2) by a maximum of plus or minus 3 pels. Two examples are given in Figure 3.12(b) and, for this mode, just the difference run-length a_1b_1 is coded. Most codewords are in this category.

3 **Horizontal mode:** This is the case when the run-length in the reference line (b_1b_2) overlaps the run-length (a_1a_2) by more than plus or minus 3 pels. Two examples are given in Figure 3.12(c) and, for this mode, the two run-lengths a_0a_1 and a_1a_2 are coded using the codewords in Figure 3.11.

more than ± 3 pels
a_0a_1 & a_1a_2 are coded using codewords

A flowchart of the coding procedure is shown in Figure 3.13. Note that the first a_0 is set to an imaginary white pel *before* the first pel of the line and hence the first a_0a_1 run-length will be $a_0a_1 - 1$. If during the coding of a line a_1, a_2, b_1, or b_2 are not detected, then they are set to an imaginary pel positioned immediately after the last pel on the respective line.

Once the first/next position of a_0 has been determined, the positions of a_1, a_2, b_1, and b_2 for the next codeword are located. The mode is then determined by computing the position of b_2 relative to a_1. If it is to the left, this indicates pass mode. If it is not to the left, then the magnitude of a_1b_1 is used to determine whether the mode is vertical or horizontal. The codeword for the identified mode is then computed and the start of the next codeword position, a_0, updated to the appropriate position. This procedure repeats alternately between white and black runs until the end of the line is reached. This is an imaginary pel positioned immediately after the last pel of the line and is assumed to have a different color from the last pel. The current coding line then becomes the new reference line and the next scanned line the new coding line.

Since the coded run-lengths relate to one of the three modes, additional codewords are used either to indicate to which mode the following codeword(s) relate – pass or horizontal – or to specify the length of the codeword directly – vertical. The additional codewords are given in a third table known as the **two-dimensional code table**. Its contents are as shown in Table 3.1. The final entry in the table, known as the **extension mode**, is a unique codeword that aborts the encoding operation prematurely before the end of the page. This is provided to allow a portion of a page to be sent in its uncompressed form or possibly with a different coding scheme.

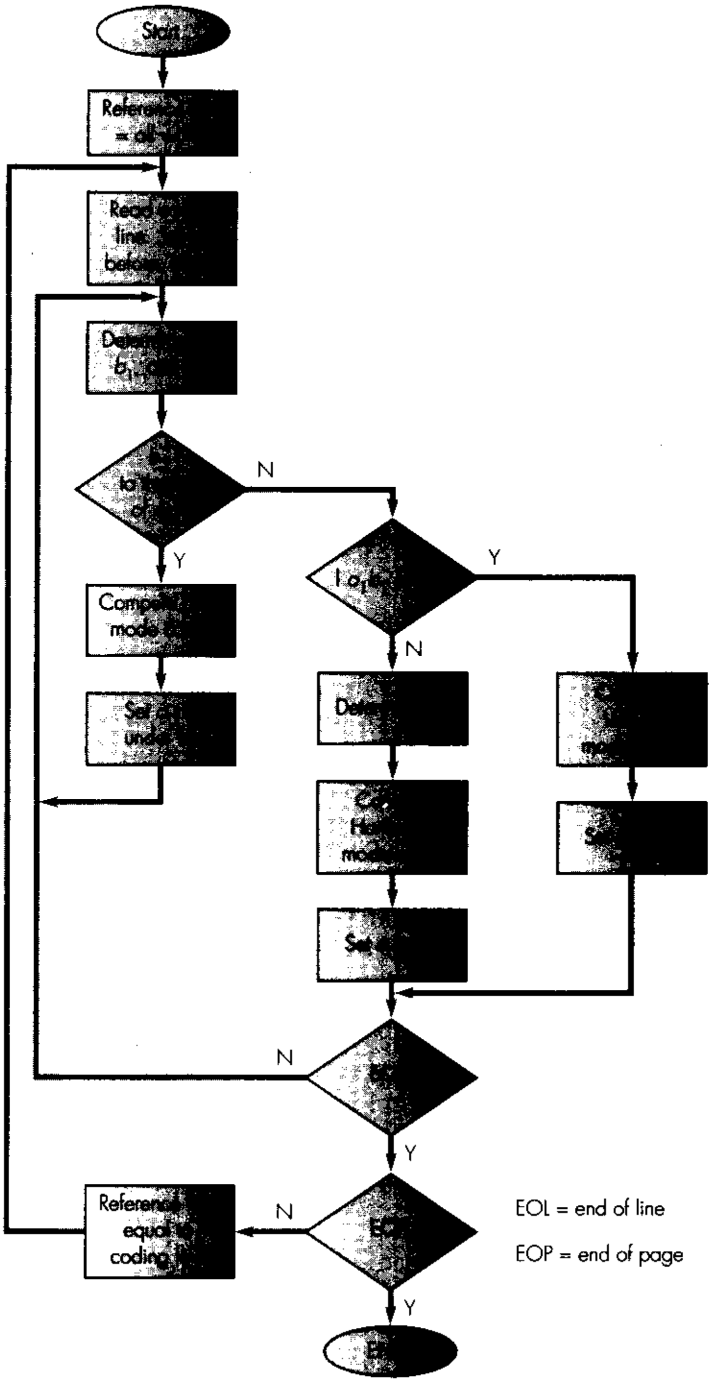


Figure 3.13 Modified-modified READ coding procedure.

Table 3.1 Two-dimensional code table contents.

Mode	Run-length to be encoded	Abbreviation	Codeword
Pass	$b_1 b_2$	P	0001 + $b_1 b_2$
Horizontal	$a_0 a_1, a_1 a_2$	H	001 + $a_0 a_1$ + $a_1 a_2$
Vertical	$a_1 b_1 = 0$	V(0)	1
	$a_1 b_1 = -1$	$V_R(1)$	011
	$a_1 b_1 = -2$	$V_R(2)$	000011
	$a_1 b_1 = -3$	$V_R(3)$	0000011
	$a_1 b_1 = +1$	$V_L(1)$	010
	$a_1 b_1 = +2$	$V_L(2)$	000010
	$a_1 b_1 = +3$	$V_L(3)$	0000010
Extension			0000001000

3.4.4 Digitized pictures

We described the digitization of both continuous-tone monochromatic pictures and color pictures in Section 2.4.3. We also calculated the amount of computer memory required to store and display these pictures on a number of popular types of display and tabulated these in Table 2.1. The amount of memory ranged from (approximately) 307 kbytes through to 2.4 Mbytes and, as we concluded, all would result in unacceptably long delays in most interactive applications that involve low bit rate networks.

In order to reduce the time to transmit digitized pictures, compression is normally applied to the two-dimensional array of pixel values that represents a digitized picture before it is transmitted over the network. The most widely-adopted standard relating to the compression of digitized pictures has been developed by an international standards body known as the **Joint Photographic Experts Group (JPEG)**. JPEG also forms the basis of most video compression algorithms and hence we shall limit our discussion of the compression of digitized pictures to describing the main principles of the JPEG standard.

3.4.5 JPEG

As we can deduce from the name, the JPEG standard was developed by a team of experts, each of whom had an in-depth knowledge of the compression of digitized pictures. They were working on behalf of the ISO, the ITU, and the IEC and JPEG is defined in the international standard **IS 10918**. In practice, the standard defines a range of different compression modes, each of which is intended for use in a particular application domain. We shall restrict our discussion here to the **lossy sequential mode** – also known as the **baseline mode** – since it is this which is intended for the compression of both monochromatic and color digitized pictures/images as used in multimedia communication applications. There are five main stages associated with this mode: image/block preparation, forward DCT, quantization, entropy encoding, and frame building. These are shown in Figure 3.14 and we shall discuss the role of each separately.

Image/block preparation

As we described in Section 2.4.3, in its pixel form, the source image/picture is made up of one or more 2-D matrices of values. In the case of a continuous-tone monochrome image, just a single 2-D matrix is required to store the set of 8-bit gray-level values that represent the image. Similarly, for a color image, if a CLUT is used just a single matrix of values is required.

Alternatively, if the image is represented as an *R*, *G*, *B* format three matrices are required, one each for the *R*, *G*, and *B* quantized values. Also, as we saw in Section 2.6.1 when we discussed the representation of a video signal, for color images the alternative form of representation known as *Y*, C_b, C_r can optionally be used. This is done to exploit the fact that the two chrominance signals, C_b and C_r, require half the bandwidth of the luminance signal, *Y*. This in turn allows the two matrices that contain the digitized chrominance components to be smaller in size than the *Y* matrix so producing a reduced form

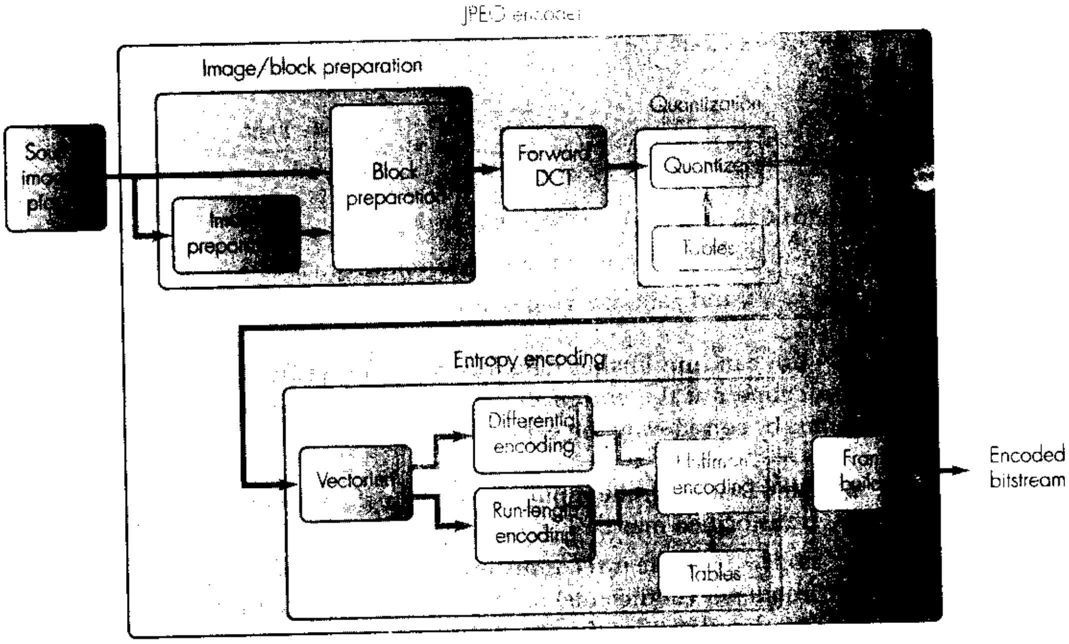


Figure 3.14 JPEG encoder schematic.

of representation over the equivalent R, G, B form of representation. For example, in the 4:2:0 format, groups of four neighboring chrominance values are averaged to produce a single value in the reduced matrix so reducing the size of the C_b and C_r matrices by a factor of four. The four alternative forms of representation are shown in Figure 3.15(a).

Once the source image format has been selected and prepared, the set of values in each matrix are compressed separately using the DCT. Before performing the DCT on each matrix, however, a second step known as **block preparation** is carried out. This is necessary since to compute the transformed value for each position in a matrix requires the values in all the locations of the matrix to be processed. It would be too time consuming to compute the DCT of the total matrix in a single step so each matrix is first divided into a set of smaller 8 × 8 submatrices. Each is known as a **block** and, as we can see in part (b) of the figure, these are then fed sequentially to the DCT which transforms each block separately.

Forward DCT

We described the principles of the DCT earlier in Section 3.2.4. Normally, each pixel value is quantized using 8 bits which produces a value in the range 0 to 255 for the intensity/luminance values – R, G, B, or Y – and a value in the range –128 to +127 for the two chrominance values – C_b and C_r. In order to compute the (forward) DCT, however, all the values are first centered around zero by subtracting 128 from each intensity/luminance value. Then, if the input 2-D matrix is represented by: $P[x, y]$ and the transformed matrix by $F[i, j]$, the DCT of each 8 × 8 block of values is computed using the expression:

each pixels — quantized using 8bits

produces a value (0-255 | RGB) Y

-128 to 127 | Cb/Cr

i/p 2-D matrix

$$F[i,j] = \frac{1}{4}C(i)C(j)\sum_{x=0}^{7}\sum_{y=0}^{7}P[x,y]\cos\frac{(2x+1)i\pi}{16}\cos\frac{(2y+1)j\pi}{16}$$

where $C(i)$ and $C(j) = 1/\sqrt{2}$ for $i, j = 0$

= 1 for all other values of i and j

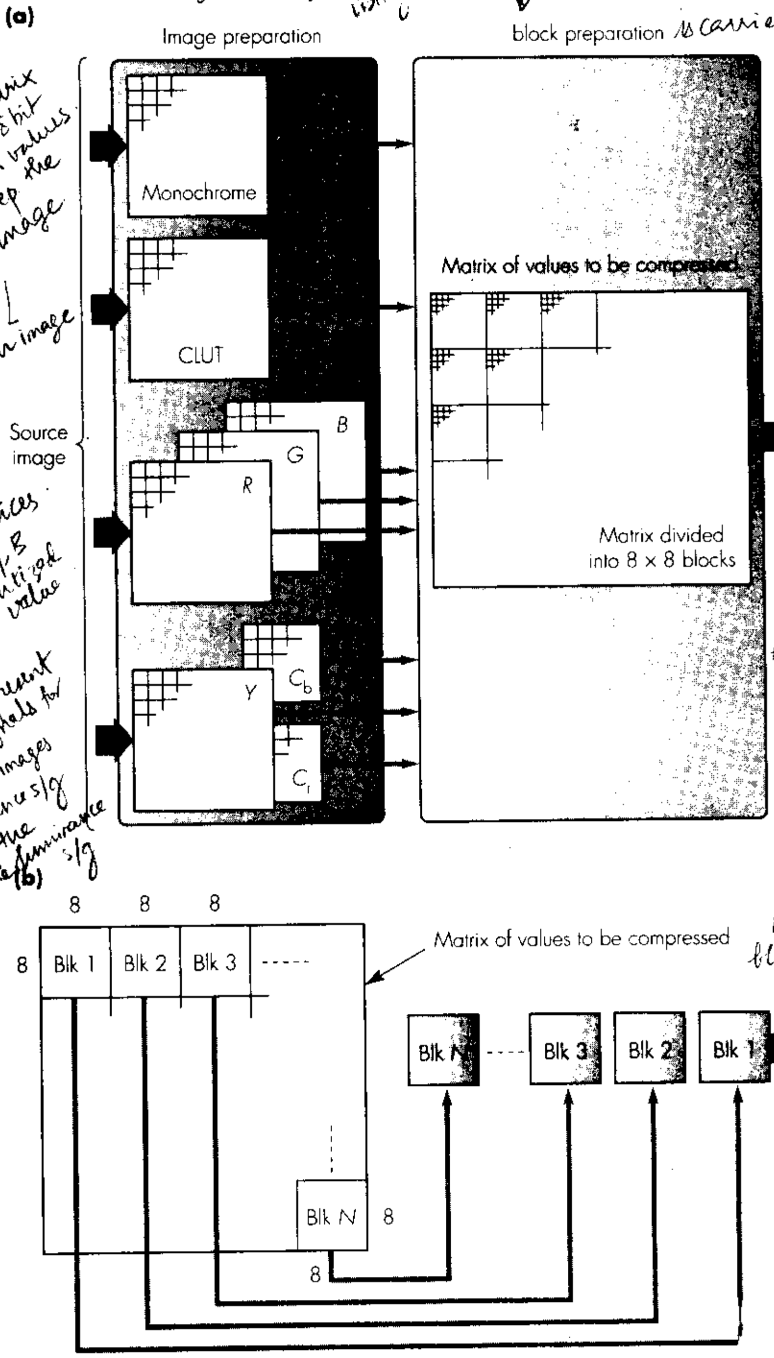
and x, y, i, and j all vary from 0 through 7.

You can find further details relating to the DCT in the bibliography for this chapter at the end of the book. However, we can deduce a number of points by considering the expression above:

- All 64 values in the input matrix, $P[x, y]$ contribute to each entry in the transformed matrix, $F[i, j]$.
- For $i = j = 0$, the two cosine terms (and hence horizontal and vertical frequency coefficients) are both 0. Also, since cos(0)=1, the value in location $F[0,0]$ of the transformed matrix is simply a function of the summation of all the values in the input matrix. Essentially, it is the mean of all 64 values in the matrix and is known as the **DC coefficient**.

64

Alternative forms [illegible]
once image fr mat selected
& prepared the set of values
in each matrix are
compressed
using DCT after
block preparation is carried out



2D matrix to store 8 bit grey level values that rep the image

colour image

3 matrices each R G, B quantized value

to represent video signals for color images
2 chrominance s/g req. half the BW of the luminance s/g

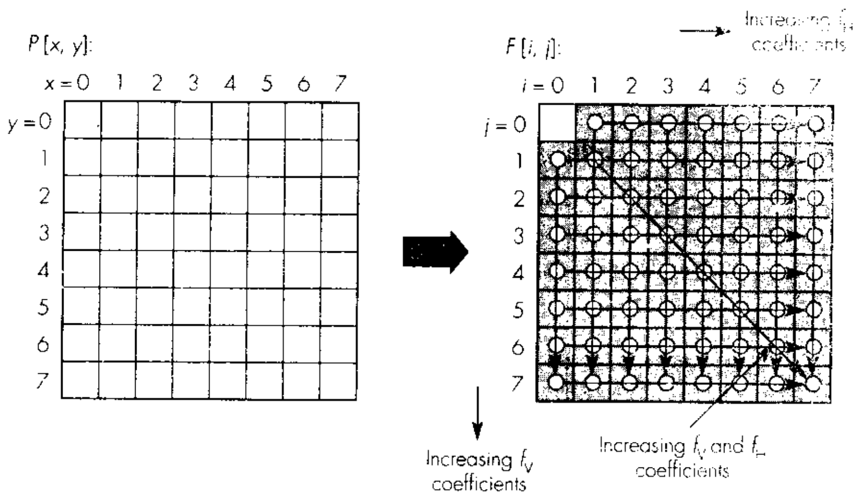
to compute txfm value for each position in the matrix reqs values in each location of the matrix to be processed

compute DCT of total DCT time consuming
8X8 matrices (blocks)
DCT txfm each block separately

Figure 3.15 Image/block preparation: (a) image preparation; (b) block preparation.

- Since the values in all the other locations of the transformed matrix have a frequency coefficient associated with them – either horizontal ($x = 1$–7 for $y = 0$), vertical ($x = 0$ for $y = 1$–7) or both ($x = 1$–7 for $y = 1$-7) – they are known as **AC coefficients**.
- For $j = 0$, only horizontal frequency coefficients are present which increase in frequency for $i = 1$–7.
- For $i = 0$, only vertical frequency coefficients are present which increase in frequency for $j = 1$–7.
- In all other locations in the transformed matrix, both horizontal and vertical frequency coefficients are present to varying degrees.

The above points are summarized in Figure 3.16. In order to gain a qualitative understanding of the likely values present in a transformed block, consider a typical image comprising, say, 640×480 pixels. Assuming a block size of 8×8 pixels, the image will comprise 80×60 or 4800 blocks each of which, for a screen width of, say, 16 inches (400 mm), will occupy a square of only 0.2×0.2 inches (5×5 mm). Hence those regions of a picture that contain a single color will generate a set of transformed blocks all of which will have firstly, the same (or very similar) DC coefficient and secondly, only a few AC


P[x, y]:
x = 0 1 2 3 4 5 6 7
y = 0 1 2 3 4 5 6 7
F[i, j]:
i = 0 1 2 3 4 5 6 7
j = 0 1 2 3 4 5 6 7
Increasing f_H coefficients
Increasing f_V coefficients
Increasing f_V and f_H coefficients

$P[x, y]$ = 8 × 8 matrix of pixel values
$F[i, j]$ = 8 × 8 matrix of transformed values/spatial frequency coefficients
In $F[i, j]$: □ = DC coefficient ▨ = AC coefficients
f_H = horizontal spatial frequency coefficient
f_V = vertical spatial frequency coefficient

Figure 3.16 DCT computation features.

coefficients within them. Thus it is only those areas of a picture which contain color transitions that will generate a set of transformed blocks with different DC coefficients and à larger number of AC coefficients within them. It is these features that are exploited in the quantization and entropy encoding phases of the compression algorithm.

Quantization

In theory, providing the forward DCT is computed to a high precision using, say, floating point arithmetic, there is very little loss of information during the DCT phase. Although in practice small losses occur owing to the use of fixed point arithmetic, the main source of information loss occurs during the quantization and entropy encoding stages where the compression takes place.

human eye

As we identified earlier in Section 3.2.4 when we first discussed transform encoding, the human eye responds primarily to the DC coefficient and the lower spatial frequency coefficients. Thus if the magnitude of a higher frequency coefficient is below a certain threshold, the eye will not detect it. This property is exploited in the quantization phase by dropping – in practice, setting to zero – those spatial frequency coefficients in the transformed matrix whose amplitudes are less than a defined threshold value. It should be noted, however, that although the eye is less sensitive to these frequency coefficients, once dropped, the same coefficients cannot be retrieved during the decoding procedure.

division op
if result is 0
↓
coeff less than threshold
non zero
↓
no. of times coeff greater than threshold

In addition to determining whether a particular spatial frequency coefficient is above a defined threshold, the quantization process aims to reduce the size of the DC and AC coefficients so that less bandwidth is required for their transmission. Instead of simply comparing each coefficient with the corresponding threshold value, a division operation is performed using the defined threshold value as the divisor. If the resulting (rounded) quotient is zero, the coefficient is less than the threshold value while if it is non-zero, this indicates the number of times the coefficient is greater than the threshold rather than its absolute value. For example, if the divisor is set to 16, then this will save 4 bits over the use of the absolute value. Clearly, this saving is at the expense of the precision used for the absolute values since in the decoder, these are determined by simply multiplying the received values by the corresponding threshold value.

threshold values
(quant. table)

As discussed, the sensitivity of the eye varies with spatial frequency, which implies that the amplitude threshold below which the eye will detect a particular spatial frequency also varies. In practice, therefore, the threshold values used vary for each of the 64 DCT coefficients. These are held in a two-dimensional matrix known as the **quantization table** with the threshold value to be used with a particular DCT coefficient in the corresponding position in the matrix.

Clearly, as we can see from the above, the choice of threshold values is important and, in practice, is a compromise between the level of compression that is required and the resulting amount of information loss that is accept-

Example 3.4

[illegible]

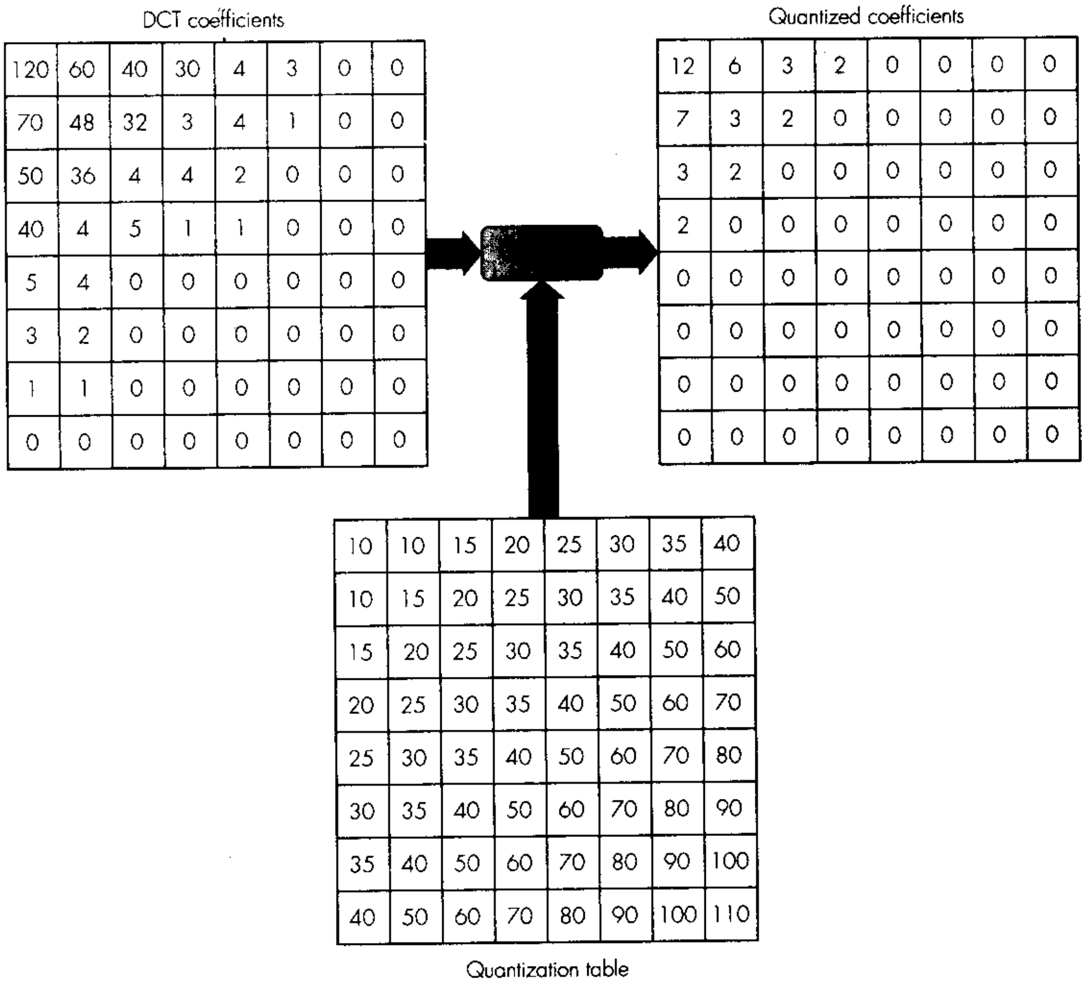
Coefficient	Quantized value	=	Rounded value	Dequantized value	Error
[illegible]	[illegible]		[illegible]	[illegible]	[illegible]

[illegible] can deduce from these [illegible] plus or minus 50% of the [illegible]

able. Although the JPEG standard includes two default quantization table values – one for use with the luminance coefficients and the other for use with the two sets of chrominance coefficients – it also allows for customized tables to be used and sent with the compressed image. An example set of threshold values is given in the quantization table shown in Figure 3.17 together with a set of DCT coefficients and their corresponding quantized values. We can conclude a number of points from the values shown in the tables:

- The computation of the quantized coefficients involves rounding the quotients to the nearest integer value.
- The threshold values used, in general, increase in magnitude with increasing spatial frequency.
- The DC coefficient in the transformed matrix is largest.
- Many of the higher-frequency coefficients are zero.

It is the last two points that are exploited during the following entropy encoding stage.



DCT coefficients

120	60	40	30	4	3	0	0
70	48	32	3	4	1	0	0
50	36	4	4	2	0	0	0
40	4	5	1	1	0	0	0
5	4	0	0	0	0	0	0
3	2	0	0	0	0	0	0
1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Quantized coefficients

12	6	3	2	0	0	0	0
7	3	2	0	0	0	0	0
3	2	0	0	0	0	0	0
2	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

10	10	15	20	25	30	35	40
10	15	20	25	30	35	40	50
15	20	25	30	35	40	50	60
20	25	30	35	40	50	60	70
25	30	35	40	50	60	70	80
30	35	40	50	60	70	80	90
35	40	50	60	70	80	90	100
40	50	60	70	80	90	100	110

Quantization table

Figure 3.17 Example computation of a set of quantized DCT coefficients.

Entropy encoding

As we saw earlier in Figure 3.14, the entropy encoding stage comprises four steps: vectoring, differential encoding, run-length encoding, and Huffman encoding. We shall describe the role of each step separately.

Vectoring The various entropy encoding algorithms we described earlier in Section 3.2.3 operate on a one-dimensional string of values, that is, a vector. As we have just seen, however, the output of the quantization stage is a 2-D matrix of values. Hence before we can apply any entropy encoding to the set of values in the matrix, we must first represent the values in the form of a single-dimension vector. This operation is known as **vectoring**.

Now
2D matrix of values → insert rep. the value in the form of a 1D vector (vectoring) → by applying entropy encoding

to exploit presence of large no of zeros zig zag scan the matrix

As we saw in Figure 3.17, the output of a typical quantization is a 2-D matrix of values/coefficients which are mainly zeros except for a number of non-zero values in the top left-hand corner of the matrix. Clearly, if we simply scanned the matrix using a line-by-line approach, then the resulting (1 × 64) vector would contain a mix of non-zero and zero values. In general, however, this type of information structure does not lend itself to compression. In order to exploit the presence of the large number of zeros in the quantized matrix, a **zig-zag scan** of the matrix is used as shown in Figure 3.18.

As we can deduce from the figure, with this type of scan, the DC coefficient and lower-frequency AC coefficients – both horizontal and vertical – are scanned first. Also, all the higher-frequency coefficients are in a sequential order so making this form of representation more suitable for compression. As we saw earlier in Figure 3.14, two different encoding schemes are applied in parallel to the values in the vector. The first is differential encoding, which is applied to the DC coefficient only, and the second is run-length encoding, which is applied to the remaining values in the vector containing the AC coefficients.

diff encoding of DC coeff

run length coding rem. vals containing AC coeff

0

Differential encoding The first element in each transformed block is the DC coefficient which is a measure of the average color/luminance/chrominance associated with the corresponding 8 × 8 block of pixel values. Hence it is the largest coefficient and, because of its importance, its resolution is kept as high as possible during the quantization phase. Because of the small physical area covered by each block, the DC coefficient varies only slowly from one block to the next.

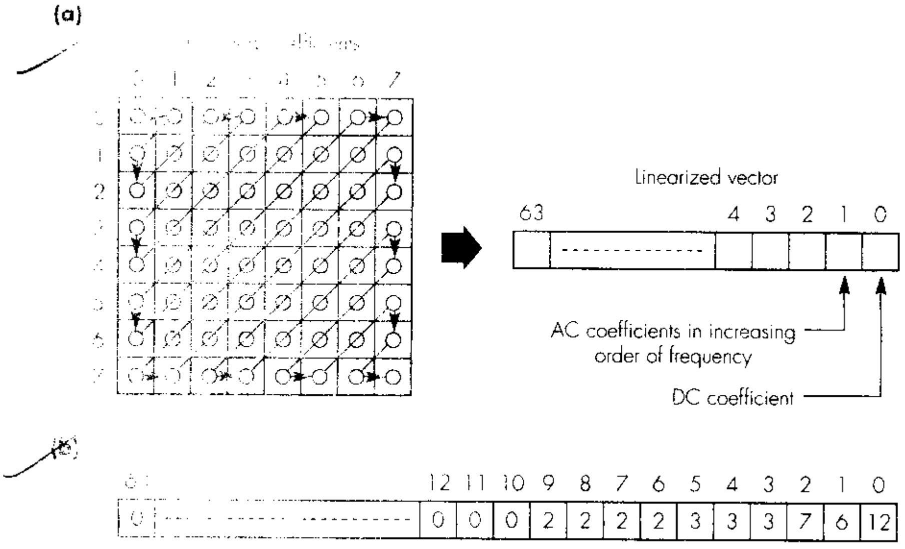


Figure 3.18 Vectoring using a zig-zag scan: (a) principle; (b) vector for example shown in Figure 3.17.

As we described in Section 3.2.4. the most efficient type of compression with this form of information structure is differential encoding ~~since this~~ encodes only the difference between each pair of values in a string rather than their absolute values. Hence in this application, only the difference in magnitude of the DC coefficient in a quantized block relative to the value in the preceding block is encoded. In this way, the number of bits required to encode the relatively large magnitudes of the DC coefficients is reduced.

For example, if the sequence of DC coefficients in consecutive quantized blocks – one per block – was:

12, 13, 11, 11, 10, ..

the corresponding difference values would be:

12, 1, −2, 0, −1, ...

the first difference value always being encoded relative to zero. The difference values are then encoded in the form (*SSS, value*) where the *SSS* field indicates the number of bits needed to encode the *value* and the value field the actual bits that represent the value. The rules used to encode each value are summarized in Figure 3.19(a).

As we can see, the number of bits required to encode each value is determined by its magnitude. A positive value is then encoded using the unsigned binary form and a negative value by the complement of this. Note also that a value of zero is encoded using a single 0 bit in the *SSS* field.

Example 3.5

Determine the encoded version of the following [illegible] which relate to the encoded DC coefficients from [illegible] blocks:

12, 1, −2, 0, −1

Answer:

Value	*SSS*	*Value*
12	4	1100
1	1	1
−2	2	01
0	0	
−1	1	0

(a)

Difference value	Number of bits needed (SSS)	Encoded value
0	0	
-1, 1	1	1 = 1 , -1 = 0
-3, -2, 2, 3	2	2 = 10 , -2 = 01 3 = 11 , -3 = 00
-7..-4, 4..7	3	4 = 100 , -4 = 011 5 = 101 , -5 = 010 6 = 110 , -6 = 001 7 = 111 , -7 = 000
-15...-8, 8...15	4	8 = 1000 , -8 = 0111 ⋮

(b)

Number of bits needed (SSS)	Huffman codeword
0	010
1	011
2	100
3	00
4	101
5	110
6	1110
7	11110
⋮	
11	111111110

Figure 3.19 Variable-length coding: (a) coding categories; (b) default Huffman codewords.

each pair
(skip, value)
no. of 0's in the run
next non zero coeff.

Run-length encoding The remaining 63 values in the vector are the AC coefficients and, because of the zig-zag scan, the vector contains long strings of zeros within it. To exploit this feature, the AC coefficients are encoded in the form of a string of pairs of values. Each pair is made up of (*skip, value*) where skip is the number of zeros in the run and *value* the next non-zero coefficient. Hence the 63 values in the vector shown earlier in Figure 3.18 would be encoded as:

(0,6) (0,7) (0,3) (0,3) (0,3) (0,2) (0,2) (0,2) (0,2) (0,0)

Note that the final pair (0,0) indicates the end of the string for this block and that all the remaining coefficients in the block are zero. Also, that the *value* field is encoded in the form *SSS/value.*

Example 3.6

Derive the binary form of the [illegible] coefficients:

[illegible]

Answer:

AC coefficients	Skip	SSS/Value
0,6	0	3 110
0,7	0	3 111
3,3	3	2 11
0,-1	0	1 0
0,0	0	0

prefix property

Huffman encoding As we saw in Section 3.4.3 when we described the encoding of digitized documents, significant levels of compression can be obtained by replacing long strings of binary digits by a string of much shorter codewords, the length of each codeword being a function of its relative frequency of occurrence. Normally, a table of codewords is used with the set of codewords precomputed using the Huffman coding algorithm. The same approach is used to encode the output of both the differential and run-length encoders.

For the differential-encoded DC coefficients in the block, the bits in the *SSS* field are not sent in their unsigned binary form as shown in Example 3.5 but in a Huffman-encoded form. This is done so that the bits in the *SSS* field have the prefix property – which we described earlier in Section 3.3.1 – and this enables the decoder to determine unambiguously the first *SSS* field from the received encoded bitstream.

Example 3.7

Determine the Huffman-encoded version of the fol[illegible] values which relate to the encoded DCT coefficients [illegible] DCT blocks.

12, 1, –2, 0, –1

Use for example purposes, the default Huffman code[illegible] lier in Figure 3.19(b).

3.7 Continued

Answer:

Value	*SSS*	*Huffman-encoded SSS*	*Encoded value*	*Encoded bitstream*
12	4	101	1100	1011100
1	1	011	1	0111
−2	2	100	01	10001
0	0	010		010
−1	1	011	0	0110

As we can deduce from the set of Huffman-encoded *SSS* fields, they illustrate that, providing the decoder uses the same set of codewords, it can readily determine the *SSS* field from the received (encoded) bitstream by searching the bitstream bit-by-bit – starting from the leftmost bit – until it reaches a valid codeword. The number of bits in the corresponding *SSS* value is then read from the table in Figure 3.19(b) and this is used to determine the number of following bits in the bitstream that contain the related *value*.

For each of the run-length encoded AC coefficients in the block, the bits that make up the *skip* and *SSS* fields are treated as a single (composite) symbol and this is then encoded using either the default table of Huffman codewords shown in Table 3.2 or a table of codewords that is sent with the encoded bitstream. Again, this is done so that the string of encoded composite symbols all have the prefix property so that the decoder can interpret the received bitstream on the correct coefficient boundaries. To enable the decoder to discriminate between the *skip* and *SSS* fields, each combination of the two fields is encoded separately and the composite symbol is then replaced by the equivalent Huffman codeword.

As we can deduce from Example 3.8, to decode the received bitstream the receiver first searches the bitstream – starting at the leftmost bit – for a valid codeword and, on finding this (100), determines the corresponding *skip* (0) and *SSS* (3) fields from the Huffman table. The *SSS* field is then used to determine the number of bits in the run-length value field and, after reading and decoding these, the process repeats until the EOB codeword is received indicating that the remaining coefficients are all zero. Because of the use of variable-length codewords in the various parts of the entropy encoding stage, this is also known as the variable-length coding (VLC) stage.

Table 3.2 Default Huffman codewords for encoding AC coefficients.

Skip/SSS	Codeword	Skip/SSS	Codeword	Skip/SSS	Codeword
[illegible]	[illegible]	3/8	[illegible]	7/6	[illegible]
[illegible]	[illegible]	3/9	[illegible]	7/7	[illegible]
[illegible]	[illegible]	3/10	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	4/1	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	4/2	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	4/3	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	4/4	[illegible]	[illegible]	[illegible]
0/7	1111000	4/5	1111111110011001	8/3	1111111110110111
[illegible]	1111110110	4/6	1111111110011010	8/4	1111111110111000
[illegible]	1111111110000010	4/7	1111111110011011	8/5	1111111110111001
0/10	1111111110000011	4/8	1111111110011100	8/6	1111111110111010
[illegible]	1100	4/9	1111111110011101	8/7	1111111110111011
[illegible]	111001	4/10	1111111110011110	8/8	1111111110111100
[illegible]	[illegible]	5/1	[illegible]	8/9	1111111110111101
[illegible]	[illegible]	5/2	1111111001	8/10	1111111110111110
[illegible]	[illegible]	5/3	1111111110011111	9/1	111111000
[illegible]	[illegible]	5/4	1111111110100000	9/2	1111111110111111
[illegible]	1111111110000101	5/5	1111111110100001	9/3	1111111111000000
[illegible]	1111111110000110	5/6	1111111110100010	9/4	1111111111000001
[illegible]	1111111110000111	5/7	1111111110100011	9/5	1111111111000010
1/10	1111111110001000	5/8	1111111110100100	9/6	1111111111000011
2/1	11011	5/9	1111111110100101	9/7	1111111111000100
2/2	11111000	5/10	1111111110100110	9/8	1111111111000101
2/3	1111110111	6/1	1111011	9/9	1111111111000110
2/4	1111111110001001	6/2	1111111000	9/10	1111111111000111
2/5	1111111110001010	6/3	1111111110100111	10/1	111111001
2/6	1111111110001011	6/4	1111111110101000	10/2	1111111111001000
2/7	1111111110001100	6/5	1111111110101001	10/3	1111111111001001
2/8	1111111110001101	6/6	1111111110101010	10/4	1111111111001010
2/9	1111111110001110	6/7	1111111110101011	10/5	1111111111001011
2/10	1111111110001111	6/8	1111111110101100	10/6	1111111111001100
3/1	111010	6/9	1111111110101101	10/7	1111111111001101
3/2	111110111	6/10	1111111110101110	10/8	1111111111001110
3/3	11111110111	7/1	11111001	10/9	1111111111001111
3/4	1111111110010000	7/2	11111111001	10/10	1111111111010000
3/5	1111111110010001	7/3	1111111110101111	11/1	111111010
3/6	1111111110010010	7/4	1111111110110000	11/2	1111111111010001
3/7	1111111110010011	7/5	1111111110110001	11/3	1111111111010010

▶

Table 3.2 Continued

Skip/SSS	Codeword	Skip/SSS	Codeword	Skip/SSS	Codeword
[illegible]	1111111111010011	13/6	1111111111100111	15/7	[illegible]
[illegible]	1111111111010100	13/7	1111111111101000	15/8	[illegible]
[illegible]	1111111111010101	13/8	1111111111101001	15/9	[illegible]
[illegible]	1111111111010110	13/9	1111111111101010	15/10	[illegible]
[illegible]	1111111111010111	13/10	1111111111101011		
[illegible]	1111111111011000	14/1	111111110110		
[illegible]	1111111111011001	14/2	1111111111101100		
[illegible]	1111111010	14/3	1111111111101101		
[illegible]	1111111111011010	14/4	1111111111101110		
[illegible]	1111111111011011	14/5	1111111111101111		
[illegible]	1111111111011100	14/6	1111111111110000		
[illegible]	1111111111011101	14/7	1111111111110001		
[illegible]	1111111111011110	14/8	1111111111110010		
[illegible]	1111111111011111	14/9	1111111111110011		
[illegible]	1111111111100000	14/10	1111111111110100		
[illegible]	1111111111100001	15/0	111111110111		
[illegible]	1111111111100010	15/1	1111111111110101		
[illegible]	11111111010	15/2	1111111111110110		
[illegible]	1111111111100011	15/3	1111111111110111		
[illegible]	1111111111100100	15/4	1111111111111000		
[illegible]	1111111111100101	15/5	1111111111111001		
[illegible]	1111111111100110	15/6	1111111111111010		

EOB = end of block

Example 3.8

Derive the composite binary symbols for the following set of run-length encoded AC coefficients:

(0,6) (0,7) (3,3) (0,–1) (0,0)

Assuming the *skip* and *SSS* fields are both encoded as a composite symbol, use the Huffman codewords shown in Table 3.2 to derive the Huffman-encoded bitstream for this set of symbols.

3.8 Continued

[illegible] AC coefficients were derived earlier [illegible]

AC coefficient	Composite symbol skip	SSS	Huffman codeword	Run-length value
[illegible]	[illegible]	[illegible]	[illegible]	6 = 110
[illegible]	[illegible]	[illegible]	100	7 = 111
[illegible]	[illegible]	[illegible]	[illegible]	2 = 10
[illegible]	[illegible]	[illegible]	00	1 = 0
[illegible]	0	0	1010	

[illegible] is then derived by adding the run- [illegible]

In this example, the number of bits required to transmit the set of AC coefficients for the 8 × 8 block of pixels is 29 and, assuming 6 bits for the DC coefficient, the total for the block is 35 bits. Hence assuming each pixel value is 8 bits, the resulting compression ratio for this block is 512/35 or, approximately, 14.6:1.

Frame building

Typically, the bitstream output by a JPEG encoder – corresponding to, say, the compressed version of a printed picture – is stored in the memory of a computer ready for either integrating with other media if necessary or accessing from a remote computer. As we can see from the above, in order for the decoder in the remote computer to be able to interpret all the different fields and tables that make up the bitstream, it is necessary to delimit each field and set of table values in a defined way. The JPEG standard, therefore, also includes a definition of the structure of the total bitstream relating to a particular image/picture. This is known as a *frame* and its outline structure is shown in Figure 3.20.

The role of the **frame builder** shown earlier in Figure 3.14 is to encapsulate all the information relating to an encoded image/picture in this format and, as we can see, the structure of a frame is hierarchical. At the top level, the complete frame-plus-header is encapsulated between a *start-of-frame* and an *end-of-frame* delimiter which allows the receiver to determine the start and

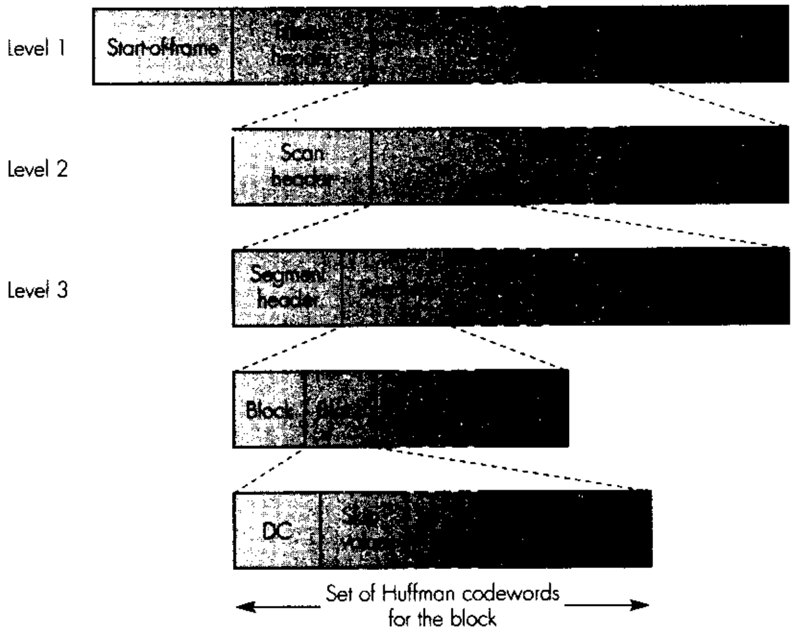


Figure 3.20 JPEG encoder output bitstream format.

end of all the information relating to a complete image/picture. The *frame header* contains a number of fields that include:

- the overall width and height of the image in pixels;
- the number and type of components that are used to represent the image (CLUT, R/G/B, $Y/C_b/C_r$);
- the digitization format used (4:2:2, 4:2:0 etc.).

At the second level, a frame consists of a number of components each of which is known as a *scan*. These are also preceded by a header which contains fields that include:

- the identity of the components (R/G/B etc.);
- the number of bits used to digitize each component;
- the quantization table of values that have been used to encode each component.

Typically, each scan/component comprises one or more *segments* each of which can contain a group of (8 × 8) blocks preceded by a header. This contains the Huffman table of values that have been used to encode each block in the segment should the default tables not be used. In this way, each segment can be decoded independently of the others which overcomes the possibility of bit errors propagating and affecting other segments. Hence each complete frame contains all the information necessary to enable the JPEG decoder to identify each field in a received frame and then perform the corresponding decoding operation.

JPEG decoding

As we can see in Figure 3.21, a JPEG decoder is made up of a number of stages which are simply the corresponding decoder sections of those used in the encoder. Hence the time to carry out the decoding function is similar to that used to perform the encoding.

On receipt of the encoded bitstream the **frame decoder** first identifies the control information and tables within the various headers. It then loads the contents of each table into the related table and passes the control information to the **image builder**. It then starts to pass the compressed bitstream to the Huffman decoder which carries out the corresponding decompression operation using either the default or the preloaded table of codewords. The two decompressed streams containing the DC and AC coefficients of each block are then passed to the differential and run-length decoders respectively. The resulting matrix of values is then dequantized using either the default or the preloaded values in the quantization table.

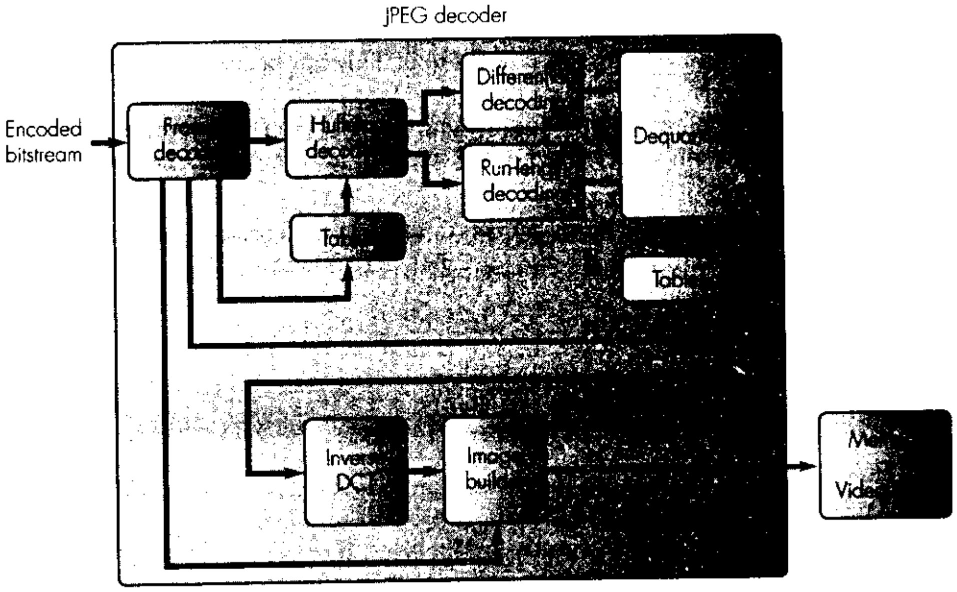


Figure 3.21 JPEG decoder schematic.

Each resulting block of 8 × 8 spatial frequency coefficients is passed in turn to the **inverse DCT** which transforms them back into their spatial form using the expression:

$$P[x, y] = \frac{1}{4}\sum_{i=0}^{7}\sum_{j=0}^{7} C(i)\,C(j)\,F[i, j]\cos\frac{(2x+1)i\pi}{16}\cos\frac{(2y+1)j\pi}{16}$$

where $C(i)$ and $C(j) = 1/\sqrt{2}$ for $i, j = 0$
$= 1$ for all other values of i and j.

The image builder then reconstructs the original image from these blocks using the control information passed to it by the frame decoder. Although the JPEG standard is relatively complicated owing to the number of encoding/decoding stages associated with it, compression ratios in excess of 20:1 can be obtained while still retaining a good quality output image/picture. This level of compression, however, applies to pictures whose content is relatively simple – that is, have relatively few color transitions – and, for more complicated pictures, compression ratios nearer to 10:1 are more common. These figures, however, assume each pixel location has three planes associated with it – R/G/B or $Y/C_b/C_r$ – and hence if a CLUT is used, then both figures can be multiplied by a factor of 3. Nevertheless, even with a compression ratio of 10:1, the amount of memory required with the various types of display tabulated in Table 2.1 is reduced to a range of from 30 kbytes through to 240 kbytes. More importantly, the time delay incurred in accessing these images is reduced by a factor of 10.

Finally, as with the GIF, it is also possible to encode and rebuild the image in a progressive way by first sending an outline of the image and then progressively adding more detail to it. This can be achieved in the following ways:

- **progressive mode:** in this mode, first the DC and low-frequency coefficients of each block are sent and then the higher-frequency coefficients;
- **hierarchial mode:** in this mode, the total image is first sent using a low resolution – for example 320×240 – then at a higher resolution such as 640×480.

Summary

In this chapter we have described a selection of the compression algorithms that are used for the compression of text and images. In general, compression is applied to both media types in order to reduce the time taken to transfer the source information over a network. This is done either to reduce the cost of the network connection or, in interactive applications, to reduce the response time to a request for the source information.

The basic techniques associated with all compression algorithms were first identified and described. These were classified as being either entropy encoding algorithms or source encoding. Entropy encoding exploits how the source information is represented and we described two examples: run-length encoding and statistical encoding.

Run-length encoding is used when the source information contains long strings of the same symbol such as a character, a bit, or a byte. Instead of sending the source information in the form of independent codewords, it is sent by simply indicating the particular symbol in each string together with an indication of the number of symbols in the string.

Statistical encoding exploits the fact that not all symbols in the source information occur with equal probability. Hence, instead of encoding all the symbols with fixed-length codewords, variable-length codewords are used with the shortest ones used to encode those symbols that occur most frequently.

In contrast, source encoding exploits a particular property of the source information in order to produce an alternative form of representation that is either a compressed version of the original form or is more amenable to the application of compression. Two examples were described: differential encoding and transform encoding.

Differential encoding is used when the amplitude of the values that make up the source information cover a large range but the difference between successive values is relatively small. Instead of using a set of relatively large codewords to represent the actual amplitude of each value, a set of smaller codewords is used, each of which indicates only the difference in amplitude between the current value being encoded and the immediately preceding value.

As the name implies, transform encoding involves transforming the source information into an alternative form of representation that lends itself more readily to the application of compression. The example that we described was the discrete cosine transform (DCT). This is used for image compression and transforms the matrix of pixel values that represent the image into a matrix of spatial frequency components which, in turn, lends itself more readily to the application of compression.

In terms of text compression, when the compressed source information is decompressed by the receiver, there is normally no loss of information. The compression algorithms that have this property are known as lossless and we described a number of such algorithms. These included both static and dynamic Huffman coding, arithmetic coding, and the LZW coding algorithm. The two Huffman algorithms and arithmetic coding are based on the relative frequency of occurrence of single characters in the source information and the LZW algorithm strings of characters.

In terms of image compression, we described a number of the algorithms that are used for the compression of graphical images, digitized documents, and digitized pictures. For use with graphical images we described the GIF and TIFF standards, for digitized documents two modified Huffman coding algorithms, and for digitized pictures the JPEG algorithm. All the algorithms that were described are part of international standards which, in addition to the compression algorithm, also define the format of the compressed information when it is being stored or transferred across a network.

Exercises

Section 3.2

3.1 Explain the meaning of the following terms relating to compression:
(i) source encoders and destination decoders,
(ii) lossless and lossy compression,
(iii) entropy encoding,
(iv) source encoding.

3.2 Explain the meaning of the following terms relating to statistical encoding:
(i) run-length encoding,
(ii) statistical encoding.

3.3 Explain the meaning of the following terms relating to statistical encoding:
(i) prefix property,
(ii) entropy,
(iii) Shannon's formula,
(iv) coding efficiency.

3.4 Explain the meaning of the following terms relating to source encoding:
(i) differential encoding,
(ii) transform encoding

3.5 With the aid of diagrams, explain in a qualitative way the meaning of the following terms relating to transform encoding:
(i) spatial frequency,
(ii) horizontal and vertical frequency components,
(iii) discrete cosine transform (DCT).

Section 3.3

3.6 Explain the meaning of the following terms relating to text compression algorithms:
(i) static coding,
(ii) dynamic/adaptive coding.

3.7 With the aid of an example, describe the rules that are followed to construct the Huffman code tree for a transmitted character set.

3.8 Messages comprising seven different characters, A through G, are to be transmitted over a data link. Analysis has shown that the relative frequency of occurrence of each character is:

A 0.10, B 0.25, C 0.05, D 0.32, E 0.01,
F 0.07, G 0.2

(i) Derive the entropy of the messages.
(ii) Use static Huffman coding to derive a suitable set of codewords.
(iii) Derive the average number of bits per codeword for your codeword set to transmit a message and compare this with both the fixed-length binary and ASCII codewords.

3.9 (i) State the prefix property of Huffman codes and hence show that your codeword set derived in Exercise 3.8 satisfied this.
(ii) Derive a flowchart for an algorithm to decode a received bit string encoded using your codeword set.
(iii) Give an example of the decoding operation assuming the received bit string comprises a mix of the seven characters.

3.10 The following character string is to be transmitted using Huffman coding:

ABACADABACADABACABAB

(i) Derive the Huffman code tree.
(ii) Determine the savings in transmission bandwidth over normal ASCII and binary coding.

3.11 With reference to the example shown in Figure 3.6 relating to dynamic Huffman coding:
(i) Write down the actual transmitted bit pattern corresponding to the character string:

"This is"

assuming ASCII coding is being used.
(ii) Deduce the extensions to the existing Huffman tree if the next word transmitted is "the".

3.12 Assuming the character set and associate probability assignments given in Figure 3.7, derive the codeword value for the character string *newt*. Assuming this is recieved by the destination, explain how the decoder determines the original string from the received codeword value.

3.13 Explain the principle of operation of the LZ compression algorithm. Hence assuming a dictionary of 16 000 words and an average word length of 5 bits, derive the average compression ratio that is achieved relative to using 7-bit ASCII codewords.

3.14 Explain the principle of operation of the LZW compression algorithm and how this is different from the LZ algorithm.

3.15 Assume the contents of a file that consists of 256 different words – each composed of alphanumeric characters from the basic ASCII character set – is to be sent over a network using the LZW algorithm. If the file contents start with the string:

This is easy as it is easy ...

show the entries in the dictionary of the encoder up to this point and the string of codewords that are sent. Also show how the receiver builds up its own dictionary and determines the original file contents from this.

3.16 Assume the same message as in Exercise 3.15 but with the number of different words much larger and unknown. How can the algorithm be changed to accommodate this?

Section 3.4

3.17 Explain the basic mode of operation of GIF. Include in your explanation the size of the color table used, how each pixel value is sent, and how the receiver knows the image parameters used by this source.

3.18 In relation to GIF, explain how the LZW coding algorithm can be applied to the (compressed) image data. Include in your explanation how compression is achieved and how the receiver interprets the compressed information.

3.19 With the aid of a diagram, describe the interlaced mode of operation of GIF. Include the potential applications of this mode and how the receiver knows it is being used.

3.20 Describe the principles of TIFF and its application domains.

3.21 Explain the meaning of the following terms relating to facsimile machines:
(i) termination codes,
(ii) make-up codes,
(iii) overscanning,
(iv) the EOL code and its uses.

3.22 Discriminate between a one-dimensional coding scheme and a two-dimensional (MMR) scheme.

3.23 Given a scanned line of pels, assuming a one-dimensional coding scheme, deduce an algorithm
(i) to determine the transmitted codewords, and
(ii) to decode the received string of codewords. Use the Huffman tables in Fig. 3.11 as a guide.

3.24 With the aid of pel patterns, assuming an MMR coding scheme, explain the meaning of the following terms:
(i) pass mode,
(ii) vertical mode,
(iii) horizontal mode. Hence with the aid of the code table given in Table 3.1, deduce an algorithm to perform the encoding operation.

3.25 With the aid of a diagram, identify the five main stages associated with the baseline mode of operation of JPEG and give a brief description of the role of each stage.

3.26 With the aid of a diagram, explain how the individual 8×8 blocks of pixel values are derived by the image and block preparation stage for each of the following source image forms:
(i) monochrome/CLUT,
(ii) RGB,
(iii) Y, C_b, C_r.

In relation to the Y, C_b, C_r format, show the order in which the blocks are output.

3.27 With the aid of a diagram, explain the meaning of the following terms relating to the DCT algorithm:

(i) DC coefficient,

(ii) horizontal and vertical spatial frequency coefficients.

Hence by considering a typical image of 1024 × 768 pixels displayed on a 17 inch (432 mm) screen, explain where the savings in bandwidth arise with JPEG.

3.28 State the characteristic of the eye that is exploited in the quantization phase of the JPEG algorithm. Hence assuming the set of DC coefficients and threshold values shown in Figure 3.17, explain how the set of quantization coefficients are derived.

3.29 Use a range of DCT coefficients and a selected quantization threshold to derive how the maximum quantization error is determined by the choice of threshold value.

3.30 State the characteristics of the values in the quantized coefficient matrix that are exploited during the entropy encoding stage. Why is vectoring using a zig-zag scan applied to the matrix?

3.31 Explain why differential encoding is used for the compression of the DC coefficients in successive blocks. By means of an example set of coefficients, estimate the savings in bandwidth that are achieved.

3.32 Using the set of coding categories listed in Figure 3.19(a), determine the encoded version of the following string of DC coefficients.

16, 15, 16, 14, 12, ...

3.33 Describe how the 63 quantized AC coefficients in a vector are encoded using run-length encoding. Hence derive the encoded form of the following vector of quantized AC coefficients:

6, 7, 0, 0, 0, 3, –1, 0, 0, ..., 0.

3.34 Describe how the differential-encoded DC coefficients from a string of successive blocks are set using Huffman encoding. Hence assuming the default Huffman codewords shown in Figure 3.19(b), derive the encoded bitstream for the set of differential encoded DC coefficients you derived in Exercise 3.32.

3.35 Describe how the set of run-length encoded AC coefficients for a block are sent using Huffman encoding. Hence assuming the set of default Huffman codewords listed in Table 3.2, derive the Huffman-encoded bitstream for the set of run-length encoded coefficients you derived in Exercise 3.33. How is the end of the set of encoded coefficients for a block determined?

3.36 All the information relating to a compressed image/picture generated by the various stages in the JPEG encoder is encapsulated within a single frame in such a way that the decoder can identify the individual fields that are present. Show the structure of a frame in a diagram and describe the role of the main fields in each of the headers that are used.

3.37 With the aid of Figure 3.21, explain how the various parts of the encoded frame identified in Exercise 3.36 are used to recreate the original image.

Identify the two alternate ways that can be used to recreate an image in a progressive way.