CHAPTER

Lossless Compression Methods

7

CHAPTER OUTLINE

Communicating Pictures. http://dx.doi.org/10.1016/B978-0-12-405906-1.00007-6

This chapter describes methods for coding images without loss; this means that, after compression, the input signal can be exactly reconstructed. These have two primary uses. Firstly, they are useful when it is a requirement of the application that loss of any kind is unacceptable. Examples include some medical applications, where compression artifacts could be considered to impact upon diagnoses, or perhaps in legal cases where documents are used as evidence. Secondly, and more commonly, they are used as a component in a lossy compression technique, in combination with transformation and quantization, to provide entropy-efficient representations of the resulting coefficients.

We first review the motivations and requirements for lossless compression and then focus attention on the two primary methods, Huffman coding and arithmetic coding. Huffman coding is used widely, for example in most JPEG codecs, and we show it can be combined with the DCT to yield a compressed bitstream with minimum average codeword length. We explore the properties and limitations of Huffman coding in Section 7.3 and then introduce an alternative method—arithmetic coding—in Section 7.6. This is an elegant and flexible, yet simple, method for encoding strings of symbols rather than single symbols as in Huffman coding. Finally, Section 7.7 presents some performance comparisons between Huffman and arithmetic coding.

7.1 Motivation for lossless image compression

For certain image compression applications, it is important that the decoded signal is an exact replica of the original. Such methods are known as "lossless" since there is no loss of information between the encoder and the decoder. They do not exploit psychovisual redundancy, but they do exploit statistical redundancy to achieve a bit rate close to the entropy of the source.

The arguments for using lossless compression can be summarized as follows:

- With some applications there can be uncertainty as to what image features are important and how these would be affected by lossy image coding. For example, in critical medical applications, errors in a diagnosis or interpretation could be legally attributed to artifacts introduced by the image compression process.
- In some applications it is essential that data is preserved intact—for example, text or numerical information in a computer file. In such cases, if the data is corrupted, it may lose value or malfunction.
- Lossless compression methods can provide an efficient means of representing the sparse matrices that result from quantization of transform coefficients in a lossy compression system. Sparsity is introduced through the use of decorrelation and

For colour and higher quality versions of Figures 7.9 and 7.10 please refer to the electronic version or the website.

quantization to exploit visual redundancy and lossless methods can further exploit the statistics of symbols, to achieve a bit rate close to the source entropy.

7.1.1 Applications

A good example of early lossless compression is the Morse code. In order to reduce the amount of data sent over the telegraph system, Samuel Morse, in the 19th century, developed a technique where shorter codewords were used to represent more common letters and numbers. His binary (digraph) system was based on dots and dashes with a space to delimit letters and an even longer space to indicate boundaries between words. For example, "a" and "e" are common letters while "q" and "j" occur less frequently, hence Morse used the mappings:

$$\text{e} \mapsto \cdot \quad \text{a} \mapsto \cdot - \quad \text{q} \mapsto - - - \cdot - \quad \text{j} \mapsto \cdot - - -$$

In practice, Morse signaling was effected by switching on and off a tone from an oscillator, with a longer tone representing a dash and a shorter duration for a dot. The full structure of Morse code for the letters of the English alphabet is given in Figure 7.1. Not shown are the other symbols represented such as numbers and special characters, including common pairs of letters such as "CH." Note the tree-structured representation used in the figure with left and right branches labeled as dots and dashes respectively. We will see a similar structure later when we examine Huffman codes. It should also be noted that the Morse mapping demands a space between codewords—because otherwise the reader could not differentiate between two "E"s and an "I." We will see later how Huffman encoding cleverly overcomes this limitation of Morse's variable length coding.

Example 7.1 (Morse code)
What is the Morse code for "MORSE CODE."

Solution. It can be seen from Figure 7.1 that the phrase MORSE CODE is represented by:

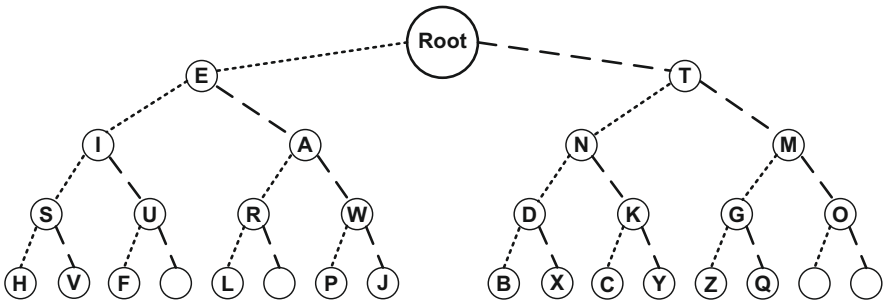


FIGURE 7.1
Symbol-code mappings for the 26 letters from the English alphabet in Morse code.

If we compare this with, let's say, a 5 bit fixed-length coding (actually there are 66 symbols in the Morse alphabet), which would require 45 bits (including a symbol for "space"), the Morse representation requires only 23 bits, a saving of approximately 50%.

Unknown to many, this encoding system is still widely used today—every radio navigation beacon used in aviation has a Morse identifier, as does every licensed airfield. Any pilot who selects such a beacon will always identify it prior to using it for navigation. This is to ensure that it is operational and that they have selected the correct beacon frequency!

7.1.2 Approaches

Several methods exist that can exploit data statistics to provide lossless compression. These include:

- **Huffman coding [1]:** This is a method for coding individual symbols at a rate close to the first order entropy, often used in conjunction with other techniques in a lossy codec.
- **Arithmetic coding [2–4]:** This is a more sophisticated method which is capable of achieving fractional bit rates for symbols, thereby providing greater compression efficiency for more common symbols.
- **Predictive coding:** This exploits data correlations, as opposed to symbol redundancies, and can be used without quantization to provide lossless image compression. As we saw in Chapter 3 and will examine further in this chapter, predictive coding is often used as a pre-processor for entropy coding.
- **Dictionary-based methods:** Algorithms such as LZ and LZW (Lempel–Ziv–Welch) [5,6] are well suited to applications where the source data statistics are unknown. They are frequently used in text and file compression.

Frequently, the above methods are used in combination. For example, DC DCT coefficients are often encoded using a combination of DPCM and either Huffman or arithmetic coding. Furthermore, as we will see in Chapter 8, motion vectors are similarly encoded using a form of predictive coding to condition the data prior to entropy coding.

Lossless encoding alone cannot normally provide the compression ratios required for most modern storage or transmission applications—we saw in Chapter 1 that requirements for compression ratios of 100:1 or even 200:1 are not uncommon. In contrast, lossless compression methods, if used in isolation, are typically restricted to ratios of around 4:1.[1]

7.1.3 Dictionary methods

Dictionary methods such as LZW (including LZ77, LZ78) are, in general, less suitable for image coding applications where the data statistics are known or can be estimated

[1] This is highly data- and application-dependent.

in advance. They do however find widespread use in more general data compression applications for text and file compression, in methods such as ZIP, UNIX compress, and GIF (Graphics Interchange Format). While these methods do not perform particularly well for natural images, methods such as PNG (Portable Network Graphics) improve upon them by pre-processing the image to exploit pixel correlations. PNG outperforms GIF and, for some images, can compete with arithmetic coding. We will not consider these techniques further here. The interested reader is referred to Sayood's book [7] for an excellent introduction.

7.2 Symbol encoding

7.2.1 A generic model for lossless compression

A generic model for lossless encoding is shown in Figure 7.2. This comprises a means of mapping the input symbols from the source into codewords. In order to provide compression, the symbol to codeword mapper must be conditioned by a representative probability model. A probability model can be derived directly from the input data being processed, from a priori assumptions or from a combination of both. It is important to note that compression is not guaranteed—if the probability model is inaccurate then no compression or even data expansion may result. As discussed in Chapter 3, the ideal symbol to code mapper will produce a codeword requiring $\log_2(1/P_i)$ bits.

7.2.2 Entropy, efficiency, and redundancy

As described in Chapter 3, H is known as the source entropy, and represents the minimum number of bits per symbol required to code the signal. For a given encoder which codes a signal at an average rate (i.e. an average codeword length) $\bar{l}$, the coding efficiency can be defined as:

$$E = \frac{H}{\bar{l}} \tag{7.1}$$

Thus, if we code a source in the most efficient manner and the code is unambiguous (uniquely decodable) then the rate of the code will equal its entropy.

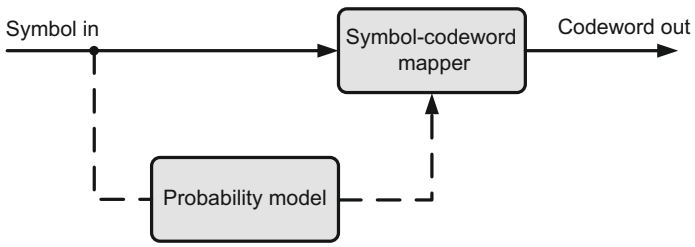


FIGURE 7.2

Generic model of lossless coding.

Table 7.1 Alternative codes for a given probability distribution.

Letter	Probability	C_1	C_2	C_3
a_1	0.5	0	0	0
a_2	0.25	0	1	10
a_3	0.125	1	00	110
a_4	0.125	10	11	111
Average length		1.125	1.25	1.75

A coding scheme can also be characterized by its redundancy, R:

$$R = \frac{\bar{l} - H}{H} \cdot 100\% \tag{7.2}$$

7.2.3 Prefix codes and unique decodability

We noted earlier that Morse code needs spaces to delineate between its variable length codewords. In modern digital compression systems we cannot afford to introduce a special codeword for "space" as it creates overhead. The way we overcome this is to use what are referred to as prefix codes. These have the property of ensuring that no codeword is the prefix of any other, and hence (in the absence of errors) that each codeword is uniquely decodable. Consider the alternative codes for the four-letter alphabet $\{a_1, a_2, a_3, a_4\}$ in Table 7.1.

Let us examine the three codes in this table. We can see that, in the case of $\mathbf{C}_1$, there is ambiguity as $a_2a_3 = a_1a_3$. Similarly for $\mathbf{C}_2$, $a_1a_1 = a_3$. In fact, only $\mathbf{C}_3$ is uniquely decodable and this is because it is a prefix code, i.e. no codeword is formed by a combination of other codewords. Because of this, it is self-synchronizing—a very useful property. The question is, how do we design self-synchronizing codes for larger alphabets? This is addressed in the next section.

Example 7.2 (Coding redundancy)
Consider a set of symbols with associated probabilities and codewords as given in the table below. Calculate the first order entropy of the alphabet and the redundancy of the encoding.

Symbol	Probability	Codeword
s_0	0.06	0110
s_1	0.23	10
s_2	0.3	00
s_3	0.15	010
s_4	0.08	111
s_5	0.06	0111
s_6	0.06	1100
s_7	0.06	1101

Solution.

1. **Average codeword length**
   The table is repeated below including the average length of each codeword. The average length $\bar{l}$ in this case is 2.71 bits/symbol.

Symbol	Probability	Code	Average length
s_0	0.06	0110	0.24
s_1	0.23	10	0.46
s_2	0.3	00	0.6
s_3	0.15	010	0.45
s_4	0.08	111	0.24
s_5	0.06	0111	0.24
s_6	0.06	1100	0.24
s_7	0.06	1101	0.24
Overall average length			**2.71**

2. **First order entropy**
   The first order entropy for this alphabet is given by:

$$\begin{aligned} H &= -\sum P \log_2 P \\ &= -(0.06 \times \log_2 0.06 + 0.023 \times \log_2 0.23 + 0.3 \\ &\quad \times \log_2 0.3 + \cdots + 0.06 \times \log_2 0.06) \\ &= 2.6849 \text{ bits/symbol} \end{aligned}$$

3. **Redundancy**
   The redundancy of this encoding is:

$$R = \frac{\bar{l} - H}{H} \times 100 = \frac{2.71 - 2.6849}{2.6849} \approx 1\%$$

7.3 Huffman coding

7.3.1 The basic algorithm

Huffman coding represents each symbol s_i with a variable length binary codeword c_i. The length of c_i is determined by rounding H_i up to the nearest integer. Huffman codes are prefix codes, i.e. no valid codeword is the prefix of any other. The decoder can thus operate without reference to previous or subsequent data and does not require explicit synchronization signals.

Huffman codes are generated using a tree-structured approach as will be described below. The decoder decodes codewords using a decoding tree of a similar form—this means that the exact tree structure must be transmitted prior to decoding or must be

Algorithm 7.1 Huffman tree formation.

1. Create ranked list of symbols $s_0 \cdots s_N$ in decreasing probability order;
2. REPEAT
3. Combine pair of symbols with lowest probabilities. i.e. those at bottom of list (indicated as s_i and s_j here);
4. Update probability of new node: $P(s_i') = P(s_i) + P(s_j)$;
5. Sort intermediate nodes: order s_i' in the ranked list;
6. UNTIL a single root node is formed with $P = 1$;
7. Label the upper path of each branch node with a binary 0 and the lower path with a 1;
8. Scan tree from root to leaves to extract Huffman codes for each symbol;
9. END.

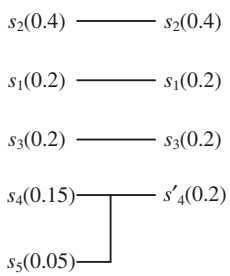
known a priori. For each symbol, the decoder starts at the tree root and uses each bit to select branches until a leaf node is reached which defines the transmitted symbol.

Huffman coding provides an efficient method for source coding provided that the symbol probabilities are all reasonably small. For coding of subband data (e.g. within JPEG) typical efficiencies between 90% and 95% are achievable. If any one of the symbol probabilities is high, then H will be small, and the efficiency of the codec is likely to be lower. Huffman coding requires that both the encoder and decoder have the same table of code words. In some systems (e.g. JPEG) this table is transmitted prior to the data (see Figure 7.7).

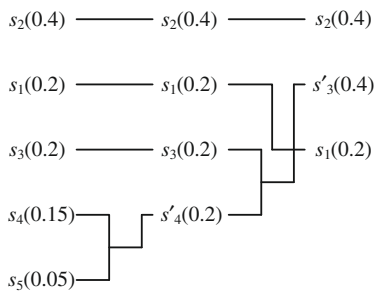
The procedure for creating a basic Huffman tree is as described by Algorithm 7.1. It should be clear that this procedure ensures that the result is a prefix code because, unlike Morse code, each symbol is represented by a unique path through the graph from root to leaf.

Example 7.3 (Basic Huffman encoding)
Consider a source which generates five symbols $\{s_1, s_2, s_3, s_4, s_5\}$ where $P(s_1) = 0.2$, $P(s_2) = 0.4$, $P(s_3) = 0.2$, $P(s_4) = 0.15$, $P(s_5) = 0.05$. Compute the Huffman tree for this alphabet.

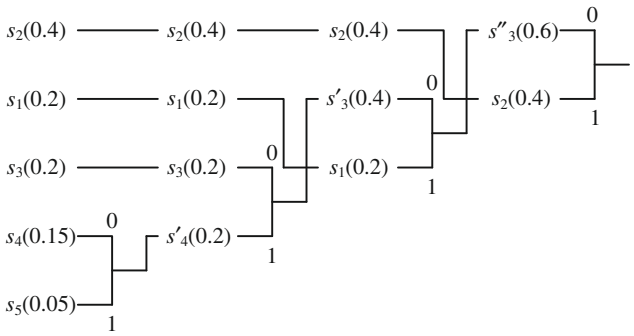
Solution. The solution is described in stages below. In the first stage the symbols are ordered in terms of their probability of occurrence and the bottom two symbols on the list are combined to form a new node $s_4'(0.2)$ where the value in parentheses is the sum of the probabilities of the two constituent nodes. We then ensure that the newly formed list of nodes is also in descending probability order as shown below:



In stage 2, we repeat this process on the newly formed list of nodes, combining the bottom two nodes to form a new node $s'_3(0.4)$. This new node is placed in the output list in a position to ensure descending probability values. This is shown below:



Finally in the third stage we form $s''_3(0.6)$ and combine this with the only other remaining node $s_2(0.4)$ to form the single root node with, as expected, a combined probability of 1:



By tracing each path through the graph, we can see that the Huffman codewords for this alphabet are:

$$s_1 = 01, \quad s_2 = 1, \quad s_3 = 000, \quad s_4 = 0010, \quad s_5 = 0011$$

This gives an average codeword length of 2.2 bits/symbol.

7.3.2 Minimum variance Huffman coding

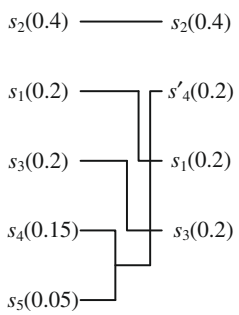
Minimum variance Huffman coding minimizes the length of the longest codeword, thus minimizing dynamic variations in transmitted bit rate. The procedure for achieving this is almost identical to the basic approach, except that when forming a new ranked list of nodes, the newly formed node is inserted in the list as high up as possible, without destroying the ranking order. This ensures that its reuse will be delayed as long as possible without destroying the ranking, thus minimizing the longest path through the graph. The revised algorithm is given in Algorithm 7.2.

Algorithm 7.2 Minimum variance Huffman tree formation.

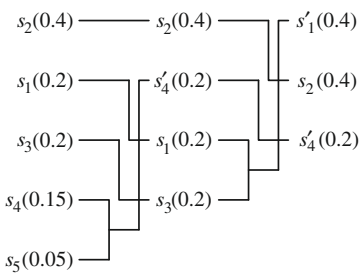
1. Create ranked list of symbols $s_0 \cdots s_N$ in decreasing probability order;
2. REPEAT
3. Combine pair of symbols with lowest probabilities, i.e. those at the bottom of the list (indicated as s_i and s_j here);
4. Update probability of new node: $P(s_i') = P(s_i) + P(s_j)$;
5. Sort intermediate nodes: insert s_i as high up as possible in the ranked list;
6. UNTIL a single root node is formed with $p = 1$;
7. Label the upper path of each branch node with a binary 0 and the lower path with a 1;
8. Scan tree from root to leaves to extract Huffman codes for each symbol;
9. END.

Example 7.4 (Minimum variance Huffman encoding)
Consider the same problem as in Example 7.3, where a source generates five symbols $\{s_1, s_2, s_3, s_4, s_5\}$ with $P(s_1) = 0.2$, $P(s_2) = 0.4$, $P(s_3) = 0.2$, $P(s_4) = 0.15$, $P(s_5) = 0.05$. Compute the minimum variance set of Huffman codes for this alphabet.

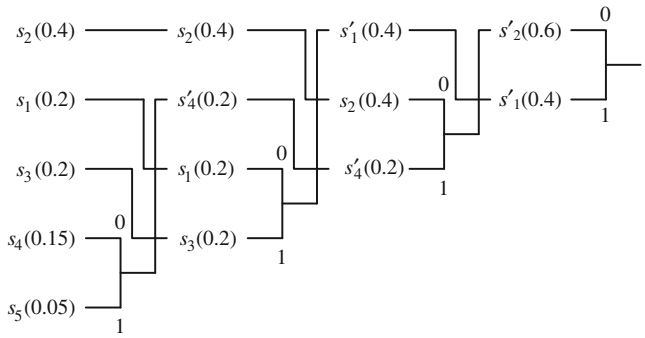
Solution. In the first stage the symbols are ordered in terms of their probability of occurrence and the bottom two symbols on the list are combined to form a new node $s_4'(0.2)$ where the value in parentheses is the sum of the probabilities of the two constituent nodes. We ensure that the newly formed node is inserted in the ordered list as high up as possible as shown below:



This process continues in a similar manner in stage 2:



Until the complete tree is formed as shown below:



By tracing each path through the graph, we can see that the Huffman codewords for this alphabet are:

$$s_1 = 10, \quad s_2 = 00, \quad s_3 = 11, \quad s_4 = 010, \quad s_5 = 011$$

This again gives an average codeword length of 2.2 bits/symbol. However, this time the minimum variance approach has reduced the maximum codeword length to 3 bits rather than the 4 bits in the previous case.

7.3.3 Huffman decoding

Huffman decoding is a simple process which takes the input bitstream and, starting at the root node, uses each bit in sequence to switch between the upper and lower paths in the graph. This continues until a leaf node is reached, when the corresponding symbol is decoded. This is described for a bitstream of K bits representing a sequence of encoded symbols, by Algorithm 7.3. For ease of understanding, this algorithm

Algorithm 7.3 Huffman decoding process.

1. INPUT bitstream representing a sequence of symbols: $b[0], b[1] \cdots b[K\text{-}1]$; INPUT Huffman tree structure;
2. i=0;
3. REPEAT
4. REPEAT
5. Start at root node;
6. IF $b[i]$=0 THEN select upper branch, ELSE select lower branch;
7. textit i=i+1;
8. UNTIL leaf node reached;
9. OUTPUT *symbol*;
10. UNTIL i=K;
11. END.

references the Huffman tree as the data structure during decoding. It should however be noted that this will normally be implemented using a state machine.

7.3.4 Modified Huffman coding

In practical situations, the alphabet for Huffman coding can be large. In such cases, construction of the codes can be laborious and the result is often not efficient, especially for biased probability distributions. For example, consider an 8 bit source with the following probabilities:

Value (x)	$P(x)$	$I(x)$	Code	Bits
0	0.2	2.232	10	2
+1	0.1	3.322	110	3
−1	0.1	3.322	1110	4
+2	0.05	4.322	11110	5
−2	0.05	4.322	11111	5
All other values	0.5	1	0 + value	9

This implies that we could generate short and efficient codewords for the first five symbols and assign an extra 1 bit flag as an escape code to indicate any of the symbols that is not one of the first five. These other values can then be fixed-length coded with (in this case) an 8 bit value. The average code length is then 2.05 bits.

An alternative approach, sometimes referred to as *extended Huffman coding*, can also be used to reduce the coding redundancy and code the alphabet at a rate closer to its entropy. Rather than coding each symbol independently, this method groups symbols together prior to code formation. As we saw in Chapter 3, this can provide significant advantages. However, there are more efficient methods for doing this such as *arithmetic coding* and we will look at these in Section 7.6.

7.3.5 Properties of Huffman codes

The Kraft inequality

An alphabet with N symbols $S = \{s_i\}$, if encoded using a binary prefix code and ranked according to their probability, P_i, will have $l_1 \leq l_2 \leq l_3 \leq \cdots \leq l_N$. The Kraft inequality states that these lengths must satisfy the following expression:

$$\sum_{i=1}^{N} 2^{-l_i} \leq 1 \tag{7.3}$$

This follows from the information content of each symbol being:

$$l_i = -\log_2 P_i = \log_2 {}^1/_{P_i}$$

In the case of a dyadic source, where all codeword lengths are integers, then:

$$\bar{l} = \sum l_i P_i = H(S)$$

Length of Huffman codes

The average length of a codeword in a Huffman encoded alphabet will depend primarily on the distribution of probabilities in the alphabet and on the number of symbols in the alphabet. We have seen above that, for integer codeword lengths, the lower bound on the average codeword length is the source entropy, H. However, in cases where the codeword length is non-integer, for practical purposes they must be rounded up to the next largest integer value. Hence:

$$\bar{l} = \sum \left\lceil \log_2 {}^1/_{P_i} \right\rceil P_i < \sum \left(\log_2 \left({}^1/_{P_i} \right) + 1 \right) P_i < H(S) + 1$$

The Huffman code for an alphabet, S, therefore has an average length $\bar{l}$ bounded by:

$$H(S) \leq \bar{l} \leq H(S) + 1 \tag{7.4}$$

Optimality of Huffman codes

In order for the code to be optimum it must satisfy the following conditions [7,8]:

1. For any two symbols s_i and s_j in an alphabet S, if $P(s_i) > P(s_j)$ then $l_i \leq l_j$ where l represents the length of the codeword.
2. The two symbols with the lowest probabilities have equal and maximum lengths.
3. In a tree corresponding to the optimum code, each intermediate node must have two branches.
4. If we collapse several leaf nodes, connected to a single intermediate node, into a single composite leaf node then the tree will remain optimal.

It can clearly be seen that Huffman codes satisfy these conditions and represent an optimum representation for coding of individual symbols. We will however see later that greater efficiencies can be obtained by encoding sequences of symbols rather than individual symbols.

7.3.6 Adaptive methods for Huffman encoding

There are two basic limitations of Huffman coding:

1. It encodes one symbol at a time.
2. It depends on a pre-computed encoding and decoding structure.

As we will see later, both of these limitations are addressed by *arithmetic coding*. However, *adaptive Huffman coding* methods have also been devised which enable dynamic updating of the Huffman tree according to prevailing source statistics. Methods include those by Gallagher [9] and more recently by Vitter [10]. The essential element, as with any adaptive system, is that the encoder and decoder models initialize

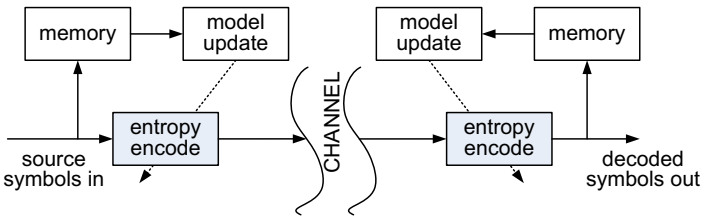


FIGURE 7.3

Adaptive entropy coding.

and update sympathetically, using identical methods. The basis for this approach is shown in Figure 7.3.

In a simple case, both encoder and decoder would initialize their trees with a single node and both would know the symbol set in use. The update process assigns a weight W to each leaf node which equals the number of times that symbol has been encountered. Symbols are initially transmitted using a predetermined set of basic codes and, as frequencies of occurrence change, the tree is updated according to an agreed procedure.

An example of this procedure is illustrated in Algorithm 7.4. We first compute the number of nodes, M, required to represent all N symbols (i.e. $2N - 1$). We label each node with a number $1 \cdots M$ and append the current W value according to the frequency of occurrence of the corresponding symbol (intermediate nodes are assigned values equal to the sum of their siblings). We define an Escape node, initially as the root node with label N and weight $W = 0$. When the first symbol s_i is encountered, it is encoded using the pre-agreed initial encoding for the alphabet. The ESC node then creates a new root node (label N) with two siblings: that for s_i labeled $N - 1$ and a new ESC node labeled $N - 2$. The new codeword for s_i is now 1 and that for ESC is 0. So when a new symbol is encountered, it must be encoded as ESC (0 currently), followed by the pre-agreed code. This process continues, spawning new nodes as new symbols are encountered.

As the frequency count of a node increments, the tree will need to be restructured to account for this. This is achieved by swapping a node with weight W with the highest numbered node with weight $W - 1$ and then rearranging nodes to provide

Algorithm 7.4 Adaptive Huffman coding: tree updating.

1. INPUT symbol s_i;
2. If symbol s_i is new then create new node with two siblings: one for s_i and a new ESC node;
3. Update weight for node s_i: $W = W + 1$;
4. Swap node with weight W with the highest numbered node with weight $W - 1$ and then rearrange nodes to provide a minimum variance structure.

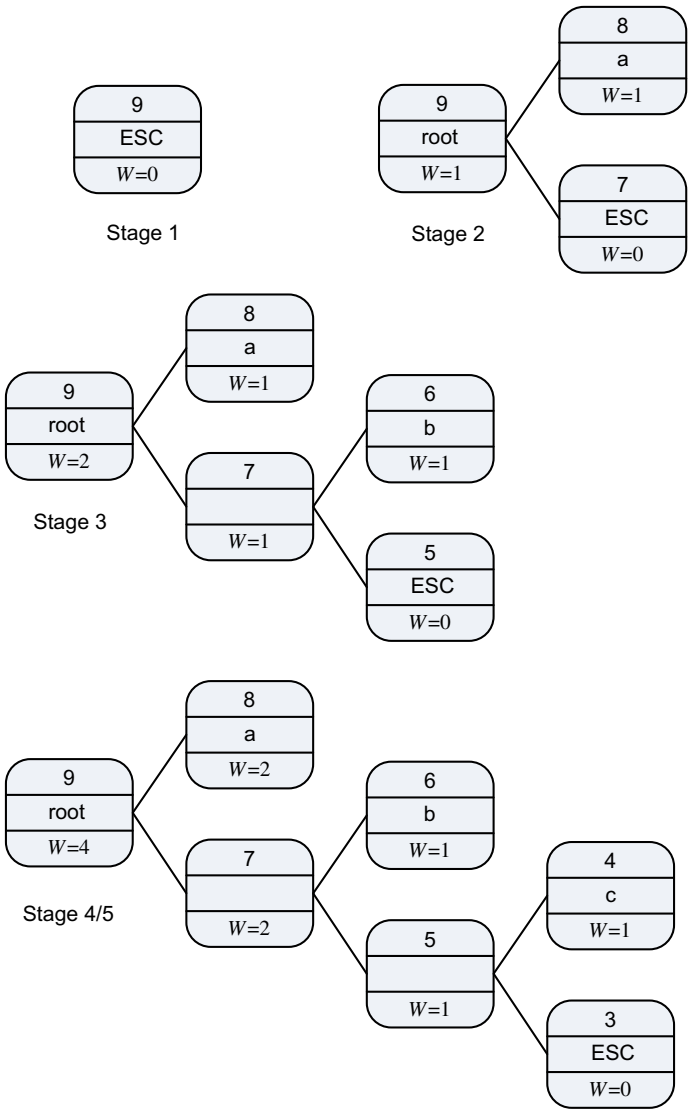


FIGURE 7.4

Example of adaptive Huffman encoding tree evolution.

a minimum variance structure. This process is capable of continually adapting to dynamic variations in symbol statistics, providing optimum codewords.

Figure 7.4 provides a simple example of the process of tree evolution for an alphabet of five symbols $\{a, b, c, d, e\}$, with the transmitted sequence $\{a, b, c, a\}$. Figure 7.5 illustrates the case of an established tree where node swapping is used to update the tree after the weight for symbol a is incremented by five counts.

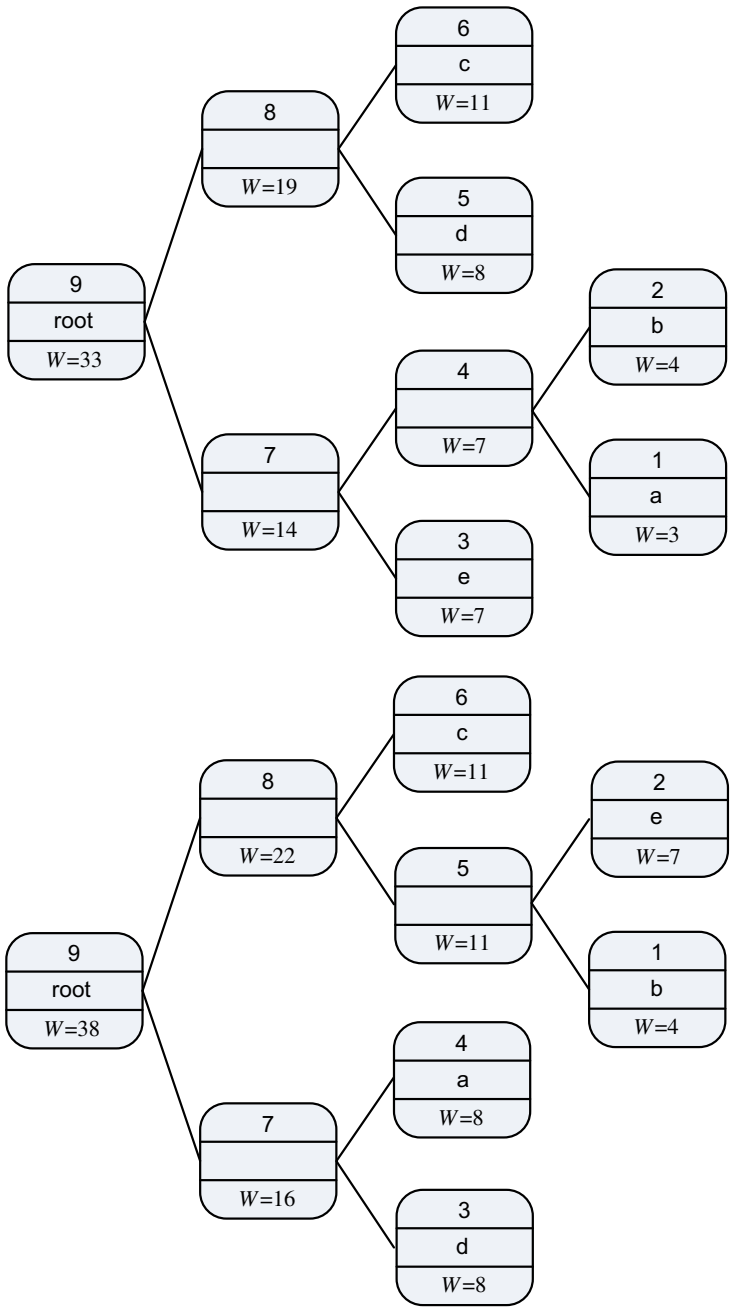


FIGURE 7.5

Example of adaptive Huffman encoding showing node swapping (bottom) after weight updating symbol *a* by five counts.

7.4 Symbol formation and encoding

7.4.1 Dealing with sparse matrices

Many data sources contain long strings of one particular symbol. For example, the quantized subband data generated by a transform or wavelet-based compression system is typically sparse. We have seen in Chapter 5 that, after applying a forward transform and quantization, the resulting matrix contains a relatively small proportion of non-zero entries with most of its energy compacted toward the lower frequencies (i.e. the top left corner of the matrix). In such cases, *run-length coding* (RLC) can be used to efficiently represent long strings of identical values by grouping them into a single symbol which codes the value and the number of repetitions. This is a simple and effective method of reducing redundancies in a sequence. A slight variant on this for sparse matrices is to code runs of zeros in combination with the value of the next non-zero entry.

In order to perform run-length encoding we need to convert the 2-D coefficient matrix into a 1-D vector and furthermore we want to do this in such a way that maximizes the runs of zeros. Consider for example the 6×6 block of data and its transform coefficients in Figure 7.6. If we scan the matrix using a zig-zag pattern, as shown in the figure, then this is more energy efficient than scanning by rows or columns.

Example 7.5 (Run-length encoding of sparse energy-compact matrices)
Consider the 4×4 quantized DCT matrix below. Perform zig-zag scanning to produce a compact 1-D vector and run-length code this using $\{run/value\}$ symbols.

$$\mathbf{C}_Q = \begin{bmatrix} 8 & 5 & 2 & 0 \\ 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Solution. Perform zig-zag scanning to produce a 1-D vector:

$$\mathbf{c}_Q = \begin{bmatrix} 8 & 5 & 3 & 0 & 0 & 2 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Next represent this vector using $\{run/value\}$ symbols:

$$\mathbf{s}_Q = \begin{bmatrix} (0/8) & (0/5) & (0/3) & (2/2) & (3/1) & \text{EOB} \end{bmatrix}$$

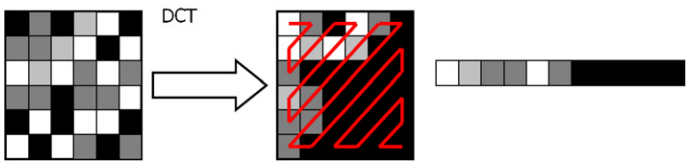


FIGURE 7.6
Zig-zag scanning prior to variable length coding.

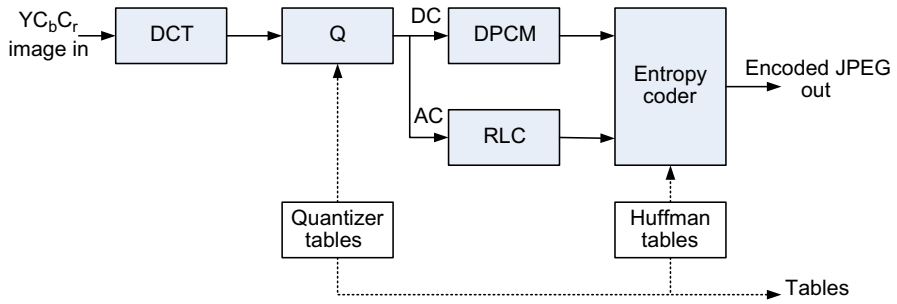


FIGURE 7.7

JPEG baseline image encoder architecture.

Note that, to avoid encoding the last run of zeros, it is normal to replace these with an End of Block (EOB) symbol.

7.4.2 Symbol encoding in JPEG

JPEG remains the universally adopted still-image coding standard. It was first introduced in 1992 and has changed little since then [11]. The operation of the baseline JPEG encoder is shown in Figure 7.7. First of all the input image, usually in 4:2:0 YC_bC_r format, is segmented into non-overlapping 8×8 blocks and these are decorrelated using an 8×8 DCT and quantized as discussed in Chapter 5. The entropy coding of these quantized coefficients is normally based on Huffman coding[2] and proceeds as follows:

- **DC coefficients** are pulse code modulated (DPCM) differentially with respect to the corresponding value from the previous block. The *size* category for the prediction residual (as defined in Table 7.4) is then Huffman encoded using a set of tables specifically designed for DC coefficients. The amplitude of the prediction residual is then appended to this codeword in ones complement form.
- **AC coefficients** are first of all zig-zag scanned as described above and then run-length encoded. The run-length symbols used for each non-zero AC coefficient are $(run/size)$, where *run* is the number of zeros preceding the next non-zero coefficient and *size* relates to the value of that coefficient, as defined in Table 7.4. These symbols are then entropy encoded using a further set of Huffman tables designed for the AC coefficients and the coefficient value is again appended to the codeword in ones complement form.

These processes are described in more detail in Algorithms 7.5 and 7.6. The approach is justified by the graph in Figure 7.8, which shows the distribution of size values

[2]It should be noted that the JPEG standard also supports arithmetic coding but this is rarely used in practice.

Algorithm 7.5 JPEG DC coefficient symbol encoding.

1. Form DPCM residual for the DC coefficient in the current block DC_i, predicted relative to the DC coefficient from the previously encoded block, DC_{i-1} : $\mathrm{DC}_{i(pred)} = \mathrm{DC}_i - \mathrm{DC}_{i-1}$;
2. Produce $symbol_{\mathrm{DC}} = (size)$ for the DC coefficient prediction;
3. Huffman encode $symbol_{\mathrm{DC}}$ using the Table for DC coefficient differences;
4. OUTPUT bits to JPEG file or bitstream;
5. OUTPUT *amplitude* value of the coefficient difference in ones complement format to the file or bitstream;
6. END.

Algorithm 7.6 JPEG AC coefficient symbol encoding.

1. Form a vector of zig-zag scanned AC coefficient values from the transformed data block as described in Figure 7.6;
2. i=0;
3. REPEAT
4. $\quad i = i + 1$;
5. $\quad$ Produce $symbol_{\mathrm{AC},i} = (run/size)$ for the next non-zero AC coefficient in the scan;
6. $\quad$ Huffman encode $symbol_{\mathrm{AC},i}$ using Table for AC coefficients;
7. $\quad$ OUTPUT codeword bits to the JPEG file or bitstream;
8. $\quad$ OUTPUT *amplitude* value of the non-zero coefficient in ones complement form to the JPEG file or bitstream;
9. UNTIL $symbol_{\mathrm{AC},i-1} = (0/0)$;
10. END.

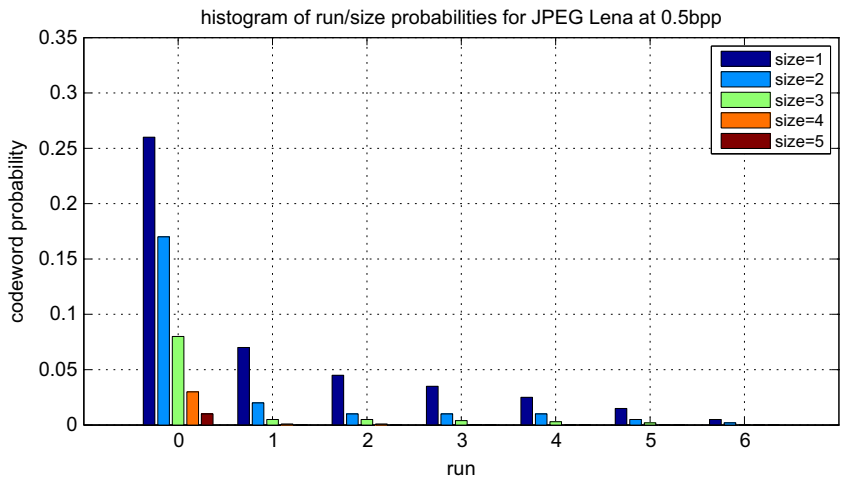


FIGURE 7.8

Distribution of *run/size* values in a typical JPEG encoding.

for various run lengths for a JPEG encoded Lena image. The figure shows statistics for 0.5 bpp but similar characteristics are obtained for other compression ratios. It can be seen that probabilities fall off rapidly as *run*-length increases and as *size* decreases. Hence longer codewords can be justified in such cases, with shorter codewords reserved for lower values of *run* and *size*.

JPEG actually restricts the number of bits allocated to run and size to 4 bits each, giving a range of 0–15 for both. If a run is greater than 15 zeros (which is unlikely) then it is divided into multiples of 15 plus the residual length. This is effective in constraining the size of the entropy code tables. Hence in JPEG, the *run*/*size* symbol is actually represented by: *RRRRSSSS*. Thus the only *run*/*size* symbols with a size of 0 are EOB (0/0) and ZRL (15/0).

Subsets of the default JPEG Huffman codes for DC and AC luminance coefficients are provided in Tables 7.2 and 7.3. A full set can be obtained from Ref. [11].

Example 7.6 (JPEG coefficient encoding)
Consider an AC luminance coefficient value of −3 preceded by five zeros. What is the JPEG symbol for this run and coefficient value? Using the default JPEG Huffman tables, compute the Huffman code for this coefficient and the resulting output bitstream.

Solution. From Table 7.4, it can be seen that the $size(-3) = 2$. The symbol is thus $s = (5/2)$.

From the default JPEG Huffman table for luminance AC coefficients, the Huffman code for (5/2) is 11111110111. The value of the non-zero coefficient is −3, which in ones complement format is 00.

The output bitstream produced for this sequence is thus:

1111111011100

Table 7.2 Table for luminance DC coefficient differences.

Category	Code length	Codeword
0	2	00
1	3	010
2	3	011
3	3	100
4	3	101
5	3	110
6	4	1110
7	5	11110
8	6	111110
9	7	1111110
10	8	11111110
11	9	111111110

Table 7.3 Table for luminance AC coefficients, showing *run*/*size* (*R*/*S*) values, Huffman codes, and their length (*L*).

R/*S*	*L*	Codeword
0/0	4	1010 (EOB)
0/1	2	00
0/2	2	01
0/3	3	100
0/4	4	1011
0/5	5	11010
0/6	7	1111000
0/7	8	11111000
0/8	10	1111110110
0/9	16	1111111110000010
0/A	16	1111111110000011
1/1	4	1100
1/2	5	11011
1/3	7	1111001
1/4	9	111110110
1/5	11	11111110110
1/6	16	1111111110000100
1/7	16	1111111110000101
1/8	16	1111111110000110
1/9	16	1111111110000111
1/A	16	1111111110001000
2/1	5	11100
2/2	8	11111001
2/3	10	1111110111
2/4	12	111111110100
2/5	16	1111111110001001
2/6	16	1111111110001010
2/7	16	1111111110001011
2/8	16	1111111110001100
2/9	16	1111111110001101
2/A	16	1111111110001110
3/1	6	111010
3/2	9	111110111
3/3	12	111111110101
3/4	16	1111111110001111
3/5	16	1111111110010000
3/6	16	1111111110010001
3/7	16	1111111110010010
3/8	16	1111111110010011
3/9	16	1111111110010100
3/A	16	1111111110010101

Example 7.7 (JPEG matrix encoding)
Consider the following quantized luminance DCT matrix. Assuming that the value of the DC coefficient in the preceding block is 40, what is the JPEG symbol sequence

Table 7.4 JPEG coefficient size categories.

Category	Amplitude range
0	0
1	$-1, +1$
2	$-3, -2, +2, +3$
3	$-7 \cdots -4, +4 \cdots +7$
4	$-15 \cdots -8, +8 \cdots +15$
5	$-31 \cdots -16, +16 \cdots +31$
6	$-63 \cdots -32, +32 \cdots +63$
etc.	etc.

for this matrix? Using the default JPEG Huffman tables, compute its Huffman codes and the resulting output bitstream. Assuming the original pixels are 8 bit values, what is the compression ratio for this encoding?

$$\mathbf{C}_Q = \begin{bmatrix} 42 & 16 & 0 & 0 & 0 & 0 & 0 & 0 \\ -21 & -15 & 0 & 0 & 0 & 0 & 0 & 0 \\ 10 & 3 & -3 & 0 & 0 & 0 & 0 & 0 \\ -2 & 2 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Solution.

1. **DC coefficient**
   The prediction residual for the current DC coefficient is $\mathrm{DC}_{i\,(\mathrm{pred})} = \mathrm{DC}_i - \mathrm{DC}_{i-1} = 2$. Thus *size* = 2 and *amplitude* = 2.
   The Huffman codeword from Table 7.2 is 011. Hence the bitstream for the DC coefficient in this case is 01110.
2. **AC coefficients**
   The zig-zag scanned vector for the matrix above is:

$$\begin{aligned} \mathbf{c}_Q = [\ & 16 \ \ -21 \ \ 10 \ \ -15 \ \ 0 \ \ 0 \ \ 0 \ \ 3 \ \ -2 \ \ 0 \ \ \cdots \\ \cdots \ & \ \ 2 \ \ -3 \ \ 0 \ \ \ \ 0 \ \ \ \ 0 \ \ 0 \ \ 0 \ \ 2 \ \ -1 \ \ 0 \ \ \cdots] \end{aligned}$$

   For each of these non-zero coefficients, we can now compute the (*run*/*size*) symbol, its Huffman code (from Table 7.3) and the corresponding ones complement amplitude representation. These are summarized in the table below.
3. **Output bitstream**
   The output bitstream for this matrix is the concatenation of DC and AC codes as follows:

$$\begin{aligned} &(011101101010000110100101010111010101100001111 1 \cdots \\ &\cdots 0111110101110111001001111111101111000010 10) \end{aligned}$$

4. **Compression ratio**
   Ignoring any header or side information, the total number of bits for this block is $5 + 82 = 87$ bits. In contrast, the original matrix has 64 entries, each of 8 bits. Hence it requires 512 bits, giving a compression ratio of 5.9:1 or alternatively 1.36 bpp.

Coefficients	Run/size	Huffman code	Amplitude	#bits
16	0/5	11010	10000	10
−21	0/5	11010	01010	10
10	0/4	1011	1010	8
−15	0/4	1011	0000	8
3	3/2	111110111	11	11
−2	0/2	01	01	4
2	1/2	11011	10	7
−3	0/2	01	00	4
2	5/2	11111110111	10	13
−1	0/1	00	0	3
EOB	0/0	1010	–	4
Total AC bits				**82**

7.4.3 JPEG performance examples

Some examples that demonstrate the performance of JPEG encoding are shown in Figures 7.9 and 7.10 for a highly textured image (*maple*) and a more structured image (*puppet*). Both original images have a wordlength of 8 bits. As can be observed, the results begin to be psychovisually acceptable beyond 0.25 bpp and become good at 1 bpp. Subjectively the result for *puppet* is more acceptable at 0.22 bpp than that for *maple*, due to the lower contribution from high frequency textures that are more heavily quantized during compression.

7.4.4 Lossless JPEG mode

A simple lossless mode is defined in the JPEG standard, which combines predictive coding with Huffman or arithmetic coding. The prediction framework is shown in Figure 7.11 and the choices of predictor are shown in Table 7.5. Predictor modes are selected from the table and signaled as side information in the header—and are normally kept constant for the whole scan. After prediction, residuals are Huffman or arithmetically encoded. In practice the reconstructed pixel values, rather than the actual pixel values, are used in the prediction to eliminate numerical mismatches between the encoder and decoder.

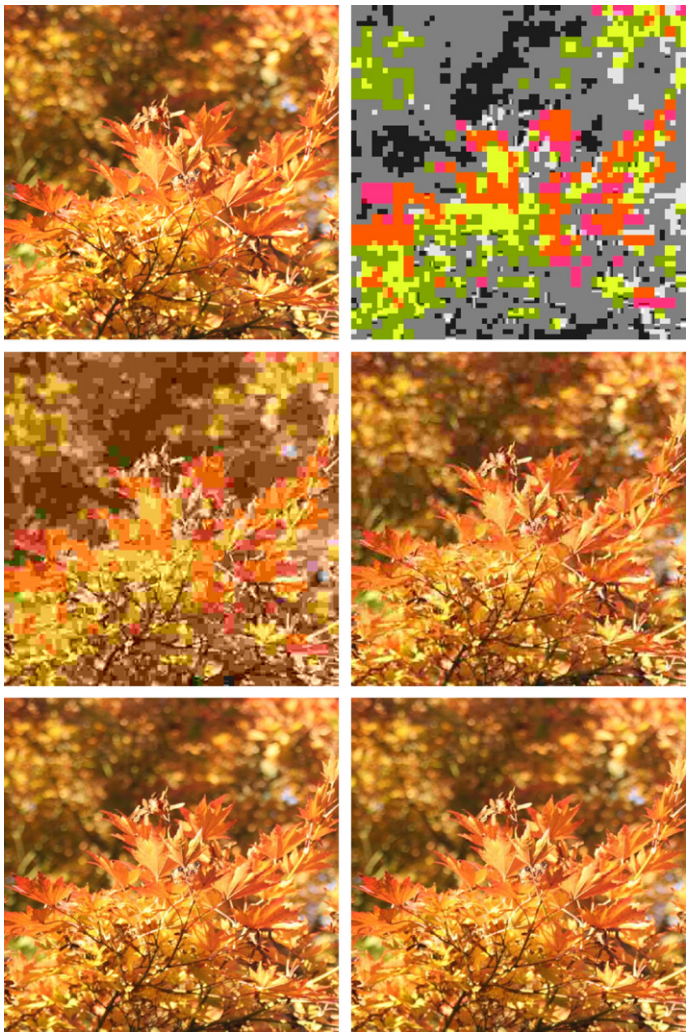


FIGURE 7.9

JPEG results for various compression ratios for 512×512 *maple* image. Top left: original. Top right: 0.16 bpp. Middle left: 0.22 bpp. Middle right: 0.5 bpp. Bottom left: 1 bpp. Bottom right: 2 bpp.

7.4.5 Context-Adaptive Variable Length Coding (CAVLC) in H.264/AVC

In its baseline and extended profiles, H.264/AVC uses two methods of entropy coding. The first, used for all syntax elements apart from transform coefficients, is based on a single codeword table for all syntax elements (SEs). The mapping of the SE to this

FIGURE 7.10

JPEG results for various compression ratios for 512×512 *puppet* image. Top left: original. Top right: 0.16 bpp. Middle left: 0.22 bpp. Middle right: 0.5 bpp. Bottom left: 1 bpp. Bottom right: 2 bpp.

table is adapted according to the data statistics. An Exp-Golomb code (see Section 7.5) is used which is effective while offering a simple decoding process.

In the case of transform coefficients, Context-Adaptive Variable Length Coding (CAVLC) is used. This provides better performance and supports a number of SE-dependent tables that can be switched between, according to the prevailing data statistics. Although CAVLC is not based on Huffman coding, it is worthy of a brief mention

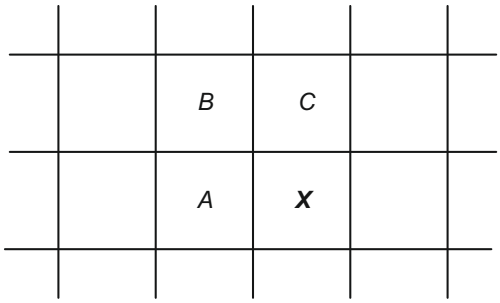


FIGURE 7.11
Prediction samples for lossless JPEG.

Table 7.5 Prediction modes for lossless JPEG.

Mode	Predictor
0	No prediction
1	$\hat{X} = A$
2	$\hat{X} = B$
3	$\hat{X} = C$
4	$\hat{X} = A + B - C$
5	$\hat{X} = A + (B - C)/2$
6	$\hat{X} = B + (A - C)/2$
7	$\hat{X} = (A + B)/2$

here since it shows the power of context switching. Good examples of CAVLC encoding are given in Ref. [12].

CAVLC exploits the sparsity of the transform matrix as we have seen previously, but also exploits other characteristics of the scanned coefficient vector, most notably: (i) that the scan often ends with a series of ± 1 values and (ii) that the numbers of coefficients in neighboring blocks are correlated. The method uses different VLC tables according to the local context.

CAVLC codes the number of non-zero coefficients and the size and position of these coefficients separately. Based on the characteristics of the data, a number of different coding modes can be invoked. For example, CAVLC can encode:

- **The number of non-zero coefficients and the *Trailing ones* (T1s):** T1s are the number of coefficients with a value of ± 1 at the end of the scan (scans often end with runs of ± 1). Four VLC tables are used for this, based on the number of coefficients in neighboring blocks.
- **The values of non-zero coefficients:** T1s can be represented as signs; other coefficients are coded in reverse scan order because variability is lower at the end of the scan. Six Exp-Golomb code tables are used for adaptation according to the magnitudes of recently coded coefficients.

- **TotalZeroes and RunBefore:** These specify the number of zeros between the start of the scan and its last non-zero coefficient. RunBefore indicates how these zeros are distributed.

A good example of CAVLC coding is provided by Richardson [13].

7.5 Golomb coding

The Golomb family includes Unary, Golomb, Golomb–Rice, and Exponential Golomb codes [14,15]. These are variable length codes which are generated via a fixed mapping between an index and a codeword. In the same way as with Huffman codes, shorter codewords can be assigned to symbols with higher probabilities and vice versa. Golomb codes have the advantage that they can be simply encoded and decoded algorithmically without the need for look-up tables. They also possess the property that, for any geometric distribution, it is possible to find an encoding that is an optimal prefix code. Adaptive methods have thus been developed and exploited in standards such as the lossless form of JPEG—JPEG-LS.

7.5.1 Unary codes

The simplest type of unary code encodes a non-negative integer, *i,* by *i* 0s followed by a single 1 (or the other way around). The first few unary codes are shown in Table 7.6 where it is assumed that the index values correspond to the alphabet symbols ranked in terms of decreasing probability. This encoding is not generally very efficient except when the probabilities are successive powers of 2.

7.5.2 Golomb and Golomb–Rice codes

Golomb codes divide all index values *i* into equal sized groups of size *m*. The codeword is then constructed from a unary code that characterizes each group, followed by a

Table 7.6 Golomb codes for index values $i = 0 \cdots 9$.

i	Unary	Golomb–Rice ($m = 4$)	Exp-Golomb
0	1	1|00	1|
1	01	1|01	01|0
2	001	1|10	01|1
3	0001	1|11	001|00
4	00001	01|00	001|01
5	000001	01|01	001|10
6	0000001	01|10	001|11
8	000000001	01|11	0001|000
9	0000000001	001|00	0001|001
⋮	⋮	⋮	⋮

fixed length code v_i that specifies the remainder of the index that has been encoded. The Golomb code residual can be simply generated using equation (7.5), where v_i represents the remainder for index i:

$$v_i = i - \left\lfloor \frac{i}{m} \right\rfloor m \tag{7.5}$$

The Golomb–Rice code is a special case where $m = 2^k$ and is shown in Table 7.6. It can be observered that Golomb–Rice codes grow more slowly than unary codes and are therefore generally more efficient.

7.5.3 Exponential Golomb codes

These were proposed by Teuhola [16] and, whereas Golomb–Rice codes divide the alphabet into equal sized groups, Exp-Golomb codes divide it into exponentially increasing group sizes. The comparative performance of Golomb–Rice and Exp-Golomb codes will depend on the size of the alphabet and the choice of m, but for larger alphabets, Exp-Golomb coding has significant advantages.

The Exp-Golomb code for a symbol index i comprises a unary code (ζ_i zeros, followed by a single 1), concatenated with a fixed length code, v_i. It can be simply generated as follows:

$$\begin{cases} v_i = 1 + i - 2^{\zeta_i} \\ \zeta_i = \lfloor \log_2(i+1) \rfloor \end{cases} \tag{7.6}$$

Exp-Golomb coding has been used in both H.264/AVC and in HEVC.

Example 7.8 (Exp-Golomb encoding and decoding)

(i) What is the Exp-Golomb code for the symbol index 228_{10}?

(ii) Show how the Exp-Golomb codeword 00000100111 would be decoded and compute the value of the corresponding symbol index.

Solution.

(i) The codeword group in this case is given by:

$$\zeta_i = \lfloor \log_2(i+1) \rfloor = 7$$

and the residual is given by:

$$v_i = 1 + i - 2^{\zeta_i} = 101_{10} = 1100101_2$$

The complete codeword is thus 000000011100101.

(ii) The algorithm is straightforward in that, for each new codeword, we count the number of leading zeros (ζ_i), ignore the following 1 and then decode the index i from the next ζ_i bits. In this case there are five leading zeros and the encoded residual $v_i = 00111_2 = 7_{10}$. The index can then be calculated from equation (7.6) as $i = v_i - 1 + 2^{\zeta_i} = 134$.

7.6 Arithmetic coding

Although Huffman coding is still widely employed in standards such as JPEG, an alternative, more flexible method has grown in popularity in recent years. Variants of arithmetic coding are employed as the preferred entropy coding mechanism in most video coding standards with *Context-based Adaptive Binary Arithmetic Coding* (CABAC) now forming the basis of H.264/AVC and HEVC [17]. Arithmetic encoding is based on the use of a cumulative probability distribution as a basis for codeword generation for a sequence of symbols. Some of the earliest references to this approach were made by Shannon [18], by Abramson in his book on information theory [2] and by Jelinek [19]. More recent popularization of the technique is due to Rissanen [3] and others. Excellent overviews can be found in Refs. [4,7,8].

7.6.1 The basic arithmetic encoding algorithm

An arithmetic encoder represents a string of input symbols as a binary fraction. In arithmetic coding, a half-open encoding range $[0, 1)$ is subdivided according to the probabilities of the symbols being encoded. The first symbol encoded will fall within a given subdivision; this subdivision forms a new interval which is then subdivided in the same way for encoding the second symbol. This process is repeated iteratively until all symbols in the sequence have been encoded.

Providing some basic additional information is known (i.e. the number of symbols transmitted or the existence of an EOB symbol), the complete sequence of symbols transmitted can thus be uniquely defined by any fractional number (referred to as a *tag*) within the range of the final symbol at the final iteration.

By using binary symbols and approximating the probabilities to powers of 2, an arithmetic encoder can be implemented using only shift and add operations. This achieves a computational complexity which is comparable to or even better than Huffman coding while providing superior compression performance. The basic arithmetic encoding algorithm is illustrated in Algorithm 7.7.

We saw in Chapter 3 that the average codeword length for a source depends highly on how symbols are created and encoded and that, by combining symbols in groups, we could potentially achieve better performance. Put simply, this is what arithmetic coding does, but it does it in a rather elegant and flexible manner.

Let us assume that we have a set of M symbols that comprise our source alphabet, $S = \{s_0, s_1, s_2, \ldots, s_{M-1}\}$, where each symbol maps to an integer x in the range $\{0, 1, 2, 3, \ldots, M-1\}$ corresponding to its symbol index. The source symbol probabilities are therefore defined as:

$$P(m) = P(x = m); \quad m = 0 \cdots M-1$$

The cumulative probability distribution can now be defined in terms of these probabilities as follows:

$$P_x(m) = \sum_{k=0}^{m} P(k); \quad m = 0 \cdots M-1 \tag{7.7}$$

Algorithm 7.7 Arithmetic encoding.

1. Subdivide the half open interval [0, 1) according to the symbol cumulative probabilities: equation (7.7);
2. Initialize upper and lower limits: $l(0)$=0; $u(0)$=1;
3. INPUT sequence of symbols $\{x_i\}$;
4. i=0;
5. REPEAT
6. i=i+1;
7. Compute lower interval limit: $l(i) = l(i-1) + (u(i-1) - l(i-1))P_x(x_i - 1)$;
8. Compute upper interval limit: $u(i) = l(i-1) + (u(i-1) - l(i-1))P_x(x_i)$;
9. UNTIL i=N;
10. OUTPUT arithmetic codeword, $v \in [l(N), u(N))$;
11. END.

Following our discussion of code optimality in Chapter 3 and in Section 7.3.5, we can see that an optimal coding system will code each symbol in S with an average length of $-\log_2 P_i$ bits.

So now we can proceed to develop the iterative symbol encoding process. Firstly we define a number line as a half open interval in the range $0 \cdots 1$, i.e. [0, 1). We can then define each accumulated probability interval at each iteration by its lower extent and its upper limit (or equivalently, its length). So, if we have a sequence of N input symbols $X = \{x_i\}$; $i = 1 \cdots N$, and we define $l(i)$ to be the lower limit at stage i of the encoding and $u(i)$ to be the corresponding upper limit, then at the start of encoding:

$$\begin{aligned} l(0) &= P(-1) = 0 \\ u(0) &= P_x(M-1) = 1 \end{aligned}$$

After the first symbol is encoded, $i = 1$:

$$\begin{aligned} l(1) &= P_x(x_1 - 1) \\ u(1) &= P_x(x_1) \end{aligned}$$

Similarly after the second symbol, $i = 2$, we have:

$$\begin{aligned} l(2) &= l(1) + (u(1) - l(1))P_x(x_2 - 1) \\ u(2) &= l(1) + (u(1) - l(1))P_x(x_2) \end{aligned}$$

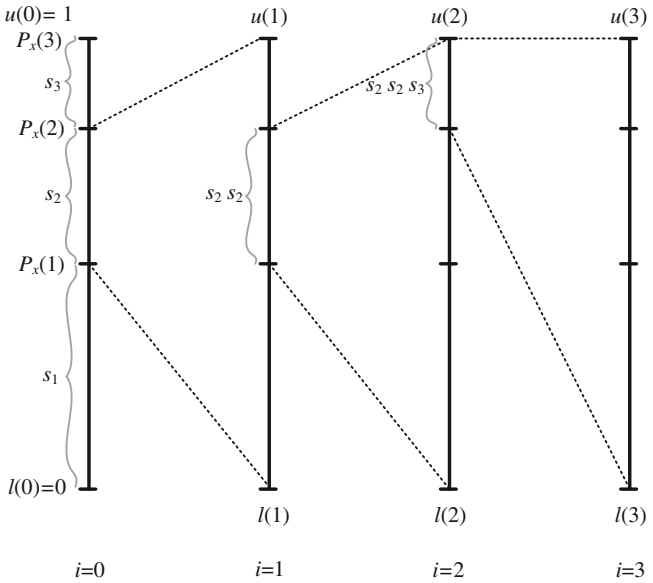
and in general after the ith symbol, x_i, we have:

$$\begin{aligned} l(i) &= l(i-1) + (u(i-1) - l(i-1))P_x(x_i - 1) \\ u(i) &= l(i-1) + (u(i-1) - l(i-1))P_x(x_i) \end{aligned} \tag{7.8}$$

This is best illustrated by a simple example (Example 7.9).

Example 7.9 (Arithmetic encoding)
Consider a sequence of symbols $\{s_2, s_2, s_3\}$ from an alphabet $\{s_1, s_2, s_3\}$ with probabilities $\{P(1), P(2), P(3)\}$. Show the arithmetic encoding process for the three iterations corresponding to this sequence.

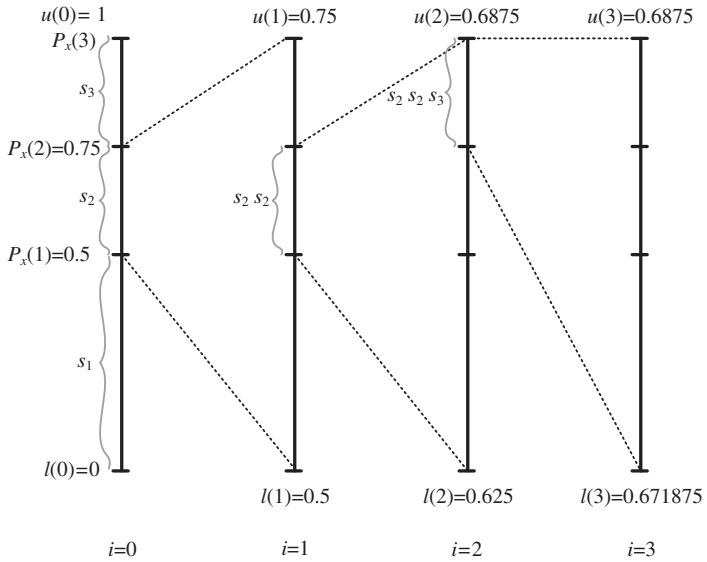
Solution. Using our notation above, the sequence maps on to $X = \{x_0, x_1, x_2\} \Rightarrow \{2, 2, 3\}$. The three iterations of the arithmetic coder for this sequence are illustrated below:



We can see that any value in the range $[l(3), u(3))$ uniquely defines this sequence provided of course that we know when to stop decoding—but more of that later.

Example 7.10 (Arithmetic encoding—numerical example)
Using the same symbols and sequence as in Example 7.9, assume that the symbol probabilities are: $P(1) = 0.5$, $P(2) = 0.25$, $P(3) = 0.25$. Compute the arithmetic code for the sequence $\{s_2, s_2, s_3\}$.

Solution. The three iterations of the arithmetic coder for this sequence are illustrated below, including the tag values for each stage:



As an example of computing the limits for the interim intervals, consider the case when $i = 2$ ($i = 1$ can be worked out by inspection):

$$
\begin{aligned}
l(2) &= l(1) + (u(1) - l(1))P_x(1) \\
l(2) &= 0.5 + (0.75 - 0.5)0.5 = 0.625 \\
u(2) &= l(1) + (u(1) - l(1))P_x(2) \\
u(2) &= 0.5 + (0.75 - 0.25)0.75 = 0.6875
\end{aligned}
$$

Repeating this for iteration 3 results in an arithmetic code that can assume any value within the interval $v \in [0.671875, 0.6875) = [0.101011_2, 0.1011_2)$. We might take the lower limit for simplicity or take a value midway between the lower and upper limits. In practice it would be preferable to use the shortest binary representation which in this case is also the lower limit, i.e. 0.101011, or, if we drop the leading zero as it is redundant, $\hat{v} = 101011$.

7.6.2 The basic arithmetic decoding algorithm

The encoding algorithm described in Section 7.6.1 creates an arithmetic codeword, $\hat{v} \in [l(N), u(N))$. The purpose of the decoder is to generate the same output sequence, $\hat{x}_i = \{x_1, x_2, \ldots, x_N\}$ as that processed at the encoder, where $\{x_i = k\} \leftarrow \{s_k\}$. We do this by recursively updating the upper and lower limits of the half open interval.

Initializing these to:

$$l(0) = P_x(0) = 0$$
$$u(0) = P_x(M-1) = 1$$

we then compute the sequence of symbol indices $\hat{x}_i$ using:

$$\hat{x}_1 = \{x : P_x(x_1 - 1) \leq \hat{v} < P_x(x_1)\} \tag{7.9}$$

So in general we need to find the next symbol index such that:

$$\hat{x}_i = \{x : l(i) \leq \hat{v} < u(i)\}; \quad i = 1, 2, \ldots, N \tag{7.10}$$

where, for each value of i we need to update the lower and upper limits:

$$\begin{aligned} u(i) &= l(i-1) + (u(i-1) - l(i-1))P_x(x_i) \\ l(i) &= l(i-1) + (u(i-1) - l(i-1))P_x(x_i - 1) \end{aligned} \tag{7.11}$$

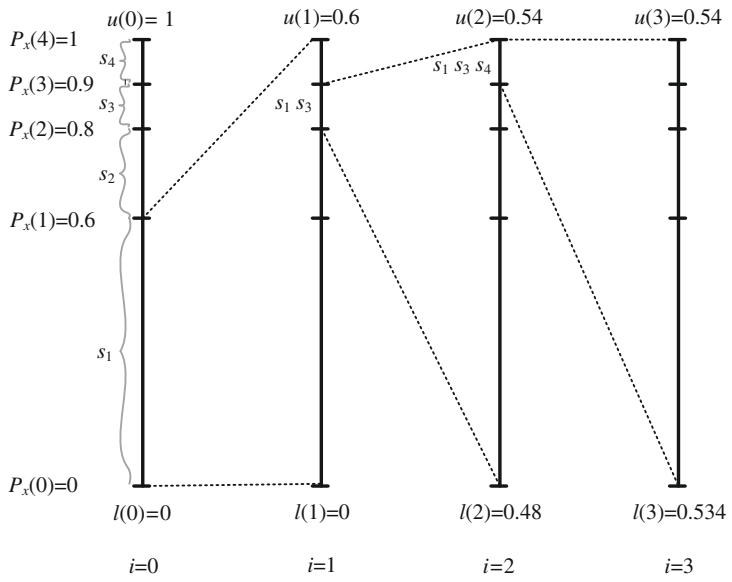
This is formalized in Algorithm 7.8.

Example 7.11 (Arithmetic decoding—numerical example)
Consider decoding the arithmetic code 0.538 which represents a three-symbol sequence from the alphabet $S = \{s_1, s_2, s_3, s_4\}$ where $P(s_1) = 0.6$, $P(s_2) = 0.2$, $P(s_3) = 0.1$, $P(s_4) = 0.1$. What is the symbol sequence represented by this arithmetic code?

Solution. The iterations of the arithmetic decoding process for this codeword are illustrated below:

Algorithm 7.8 Arithmetic decoding.

1. Initialize upper and lower limits: $l(0)$=0; $u(0)$=1;
2. INPUT arithmetic code tag, $\hat{v}$;
3. i=0;
4. REPEAT
5. i=i+1;
6. Compute symbol index i: $\hat{x}_i = \{x : l(i) \leq \hat{v} < u(i)\}$ where
   upper interval limit: $u(i) = l(i-1) + (u(i-1) - l(i-1))P_x(x_i)$ and
   lower interval limit: $l(i) = l(i-1) + (u(i-1) - l(i-1))P_x(x_i - 1)$;
7. OUTPUT symbol s_{x_i};
8. UNTIL $i = N$;
9. END.



We can see that at stage 1, for decoding symbol 1, the tag lies in the range $[0, 0.6)$ so the symbol output is s_1. Computing the modified cumulative probability values for stage 2 we see that the tag falls in the range [0.48, 0.54) corresponding to the symbol sequence $\{s_1, s_3\}$.

Repeating this for iteration 3 results in the tag falling in the interval $[0.534, 0.54)$ which represents the sequence: $\{s_1, s_3, s_4\}$.

7.6.3 Advantages of arithmetic coding

Arithmetic coding has a number of important properties that make it attractive as a preferred method for entropy encoding. These include:

1. Huffman coding can be efficient with large symbol alphabets, where the highest probability for any individual symbol $p_{\max}$ is small and the distribution is relatively uniform. However, for smaller alphabets and where the probability distribution is biased, Huffman coding can be inefficient. In arithmetic coding, codewords represent strings of symbols, hence offering the potential for fractional wordlengths and a corresponding reduction in overall bit rate.
2. Arithmetic coding separates the modeling process from the coding process. Each individual symbol can have a separate probability (usually referred to as conditioning probability) which may be either predetermined or calculated adaptively according to previous data.
3. Symbol statistics can be simply adapted to changing conditions without major recomputation of coding tables.

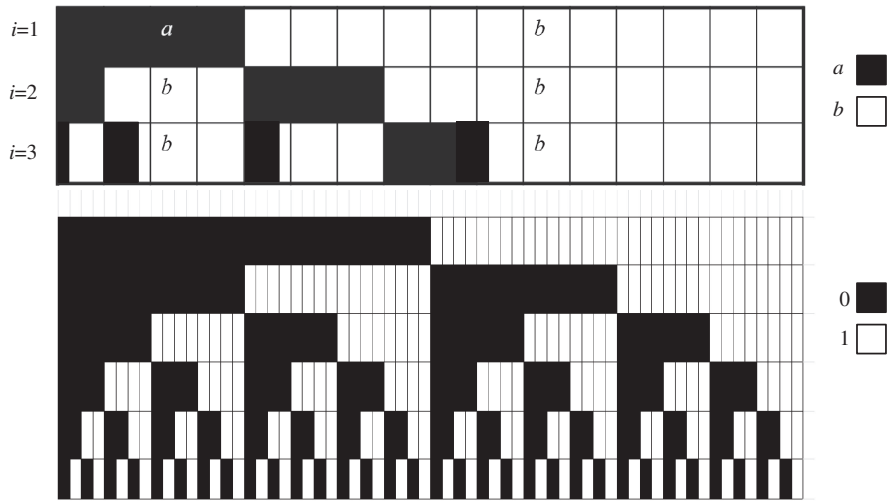


FIGURE 7.12

A graphical illustration of the arithmetic encoding and decoding processes. Top: probability intervals for three iterations. Bottom: codeword intervals. Shows $P(a) = 0.25$ and $P(b) = 0.75$ with example sequences *abb* and *bbb*. Adapted from Ref. [20].

7.6.4 Binary arithmetic coding

In practical arithmetic coding systems, binary implementations are essential. This poses some constraints on the operation in terms of finite wordlength effects and uniqueness, but also presents opportunities in terms of efficient implementations. To help understanding of this process, consider the illustration in Figure 7.12. This shows three iterations of an arithmetic encoder or decoder for the case of a system with two symbols $\{a, b\}$ where $p(a) = 0.25$ and $p(b) = 0.75$. The top diagram shows probability intervals similar to those we have seen before but visualized slightly differently. The bottom diagram shows the corresponding binary codewords. By scanning vertically from a given sequence, we can easily identify the binary codewords that uniquely define that sequence and select the codeword with the minimum length. For example, consider the sequence $\{a, b, b\}$; scanning down to the bottom diagram, we can see that the shortest codeword that uniquely identifies this sequence is 001. Similarly for $\{b, b, b\}$ it is 11.

Binary encoding can provide advantages in sequential bit-wise processing of tag values, both in encoding and decoding stages. This is demonstrated in the following example.

Example 7.12 (Binary arithmetic encoding)

Consider a source with four symbols with probabilities: $P(s_0) = 0.5$, $P(s_1) = 0.25$, $P(s_2) = 0.125$, $P(s_3) = 0.125$. Perform arithmetic encoding of the sequence: $S = \{s_1, s_0, s_2\}$.

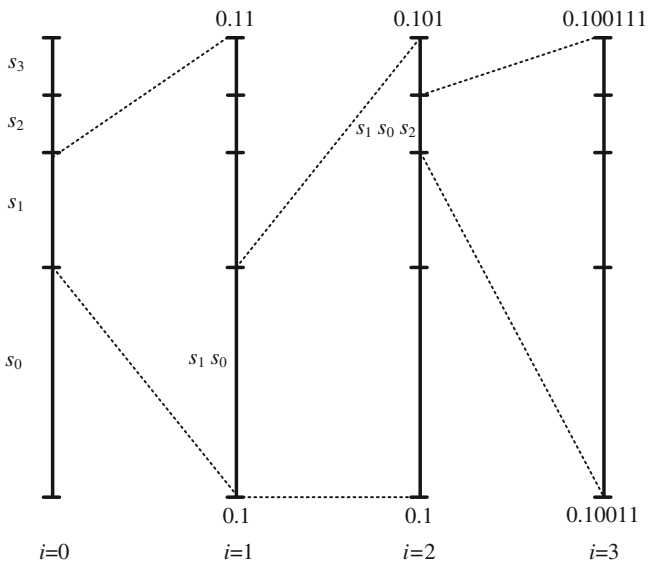
Solution. The encoding proceeds as follows:

Stage 1
The first symbol s_1 defines the interval $I_1 = [0.5, 0.75)_{10} = [0.10, 0.11)_2$. Examining the upper and lower limits of this interval we can see that the first binary digit after the binary point (1) is invariant and hence can be transmitted or appended to the output file.

Stage 2
The second symbol s_0 defines the interval $I_2 = [0.5, 0.625)_{10} = [0.100, 0.101)_2$. Examining the upper and lower limits of this interval we can see that the first two binary digits after the binary point (10) are invariant and hence the 0 can be transmitted or appended to the output file.

Stage 3
The third symbol s_2 defines the interval and arithmetic code: $I_3 = [0.59375, 0.609375)_{10} = [0.10011, 0.100111)_2$. Examining the upper and lower limits of this interval we can see that the first five binary digits after the binary point (10011) are invariant and hence can be transmitted or appended to the output file. We can transmit any value in this range as our arithmetic code, $\hat{v} \in [0.10011, 0.100111)_2$. In this case we can simply send an extra zero to differentiate the lower limit from the upper limit, i.e. $\hat{v} = 0.100110$.

A simplified diagram of this arithmetic encoding process is shown in the figure below.



Example 7.13 (Binary arithmetic decoding)
Decode the transmitted sequence from Example 7.12.

Solution. The decoding process is summarized in the following table. As each bit is loaded, it can be processed to further differentiate symbols. When the desired number of symbols or an EOB symbol is received, the decoding terminates. Consider the first row of the table: the first one does not uniquely define a symbol—it could be s_2 or s_3 in this case. However, when the second bit is processed, it uniquely defines s_1. This is repeated until the complete sequence is decoded.

Rx bit	Interval	Symbol
1	$[0.5, 1)_{10} = [0.1, 1.0)_2$	–
0	$[0.5, 0.75)_{10} = [0.10, 0.11)_2$	s_1
0	$[0.5, 0.625)_{10} = [0.100, 0.101)_2$	s_0
1	$[0.5625, 0.625)_{10} = [0.1001, 0.1100)_2$	–
1	$[0.59375, 0.625)_{10} = [0.10011, 0.10100)_2$	–
0	$[0.59375, 0.609375)_{10} = [0.100110, 0.100111)_2$	s_2

7.6.5 Tag generation with scaling

In cases where a transmitted sequence of symbols is large, the arithmetic encoding process will have to iterate a large number of times and at each iteration the size of the intervals reduces (on average by a factor of 2). To cope with this in practical situations, the wordlength must be sufficient to ensure that the intervals and tag can be represented with sufficient numerical accuracy. In practice very long wordlengths are mitigated by a process of scaling the interval each time a bit of the tag is transmitted. Consider stage 1 in Example 7.12; here $l(1) = 0.5$ and $u(1) = 0.75$. Once the first binary 1 is transmitted, this means that the codeword is forever constrained to the upper half of the number line. So, without any loss of generality, we can scale the interval by a factor of 2 as follows:

$$l'(1) = 2(l(1) - 0.5) = 0$$
$$u'(1) = 2(u(1) - 0.5) = 0.5$$

The equations for the next stage are then generated as follows:

$$\begin{aligned} u(i) &= l'(i-1) + (u'(i-1) - l'(i-1))P_x(x_i) \\ l(i) &= l'(i-1) + (u'(i-1) - l'(i-1))P_x(x_i - 1) \end{aligned} \tag{7.12}$$

This process of scaling is repeated every time a new symbol constrains the tag to be completely contained in the upper or lower half interval. In cases where the tag

interval straddles the center of the number line we can perform a similar process, but this time scaling when the interval falls within [0.25, 0.5). So we have three types of scaling or renormalization that can be used after interval subdivision [4,7]:

$$\begin{aligned} E_1 : [0, 0.5) \rightarrow [0, 1); \quad & E_1(x) = 2x \\ E_2 : [0.5, 1) \rightarrow [0, 1); \quad & E_2(x) = 2(x - 0.5) \\ E_3 : [0.25, 0.75) \rightarrow [0, 1); \quad & E_3(x) = 2(x - 0.25) \end{aligned} \tag{7.13}$$

It is not difficult to see how this process can be mimicked at the decoder as the tag bits are processed.

7.6.6 Context-adaptive arithmetic coding

Arithmetic coding has been supported in a number of standards such as JPEG, JPEG2000, H.263, H.264/AVC, and HEVC. However, not until its inclusion in H.264/AVC has it been commonly used in practice. This uptake was due to the fact that Context-Adaptive Binary Arithmetic Coding (CABAC) [17] provided significant performance gains over previous methods. CABAC is now the standard entropy encoding method in HEVC.

It has been shown that, by taking account of prevailing context and adapting conditioning probabilities accordingly, then significant improvements can be obtained with entropy coders. We saw an example of how to do this with Huffman coding earlier in this chapter and also looked at CAVLC. We will consider it briefly here in the context of arithmetic coding. A good review of adaptive coding methods is provided by Tian et al. [12].

Several realizations of adaptive arithmetic encoding engines have been reported in the literature. These include the Q Coder [21], which exploits the idea of renormalization after subdivision (as discussed above) to avoid numerical precision issues (an extension of this, known as the QM Coder, is used in JPEG), and the MQ coder [22]. The latter is used for encoding bit-planes of quantized wavelet coefficient codeblocks in the JPEG2000 EBCOT (Embedded Block Coding with Optimized Truncation) coder.

CABAC has been adopted for the Main and High Profiles of H.264/AVC and HEVC, and employs context-based adaptivity to improve performance. This enables the encode to dynamically adapt to prevailing symbol statistics. In CABAC, conditioning probabilities are based on the statistics of previously encoded Syntax Elements, which are used to switch between a number of probability models. It offers a fast and accurate means of estimating conditioning probabilities over a short interval and excellent computational efficiency.

CABAC comprises three main stages: binarization, context modeling, and binary arithmetic coding. These are illustrated in Figure 7.13. These steps are briefly described below. The reader is referred to Refs. [12,17] for a more detailed operational explanation.

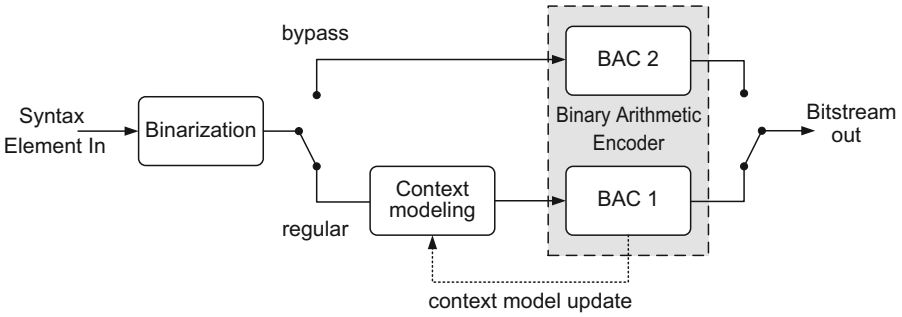


FIGURE 7.13
Context-Adaptive Binary Arithmetic Coding. Adapted from Ref. [17].

Binarization: Maps non-binary valued SEs to a bin string (a sequence of binary decisions). This reduces the symbol alphabet by producing a set of intermediate codewords (bin string) for each non-binary SE. It also enables context modeling at a subsymbol level, where conditional probabilities can be used for more common bins and simpler schemes for less frequent bins. In CABAC, binarization is implemented using a number of schemes and combinations of them. These include: Unary and Truncated Unary, Fixed Length, *k*th order Exp-Golomb and combinations of these.

Context modeling: We have seen earlier that one of the primary advantages of arithmetic coding is its ability to separate modeling from coding. CABAC exploits this. The modeler assigns a probability distribution to the symbols being processed and this is used to condition the BAC. In CABAC, the context modeling is restricted to a small number of local symbols, in order to provide a good trade-off between modeling complexity and performance. In cases where SEs exhibit uniform or close to uniform probability distributions, then a bypass mode is invoked.

Binary arithmetic coding: CABAC employs a multiplication-free approach to arithmetic encoding and uses recursive interval subdivision and renormalization based on the approach of Witten [23].

In the H.264/AVC and HEVC standards, CABAC is used to encode coding parameters (e.g. macroblock type, spatial and temporal prediction mode, and slice and macroblock control information) as well as prediction residuals. CABAC has been reported to offer up to 20% improvement over CAVLC, with typical savings between 9% and 14% [17].

7.7 Performance comparisons—Huffman vs arithmetic coding

A comparison of the compression performance achieved for basic arithmetic and Huffman coding is shown in Figure 7.14. These results represent the combination of a DCT codec with both types of entropy coder. It shows that arithmetic coding offers

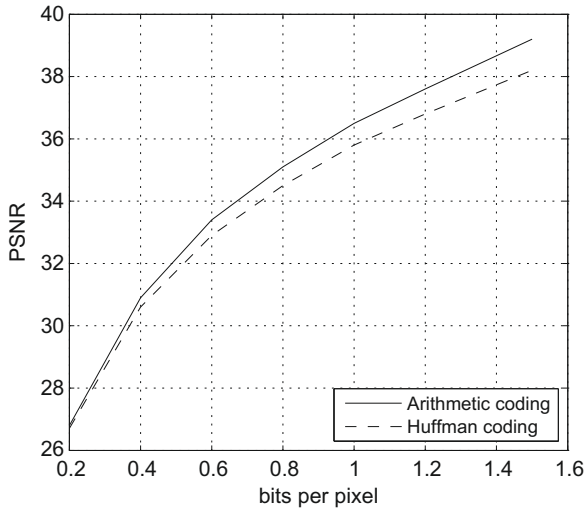


FIGURE 7.14
Huffman vs arithmetic coding.

advantages (up to 1 dB at lower compression ratios) compared with Huffman coding. As we have seen, further improvements are possible using more sophisticated adaptive and context-based techniques which suit arithmetic coding better than Huffman coding.

7.8 Summary

This chapter has introduced methods for coding images without loss. We have seen the requirement that useful codes need to have prefix properties so as to avoid the need for explicit synchronization. Huffman codes satisfy this property and are able to code individual symbols from a source at a rate close to its entropy. We then described arithmetic coding which offers performance advantages in that it can represent multiple symbols in a group and hence offer fractional symbol wordlengths. It is also more amenable to adaptive encoding—supporting dynamic modification of conditioning probabilities.

References

[1] D.A. Huffman, A method for the construction of minimum-redundancy codes, Proceedings of the IRE 40 (9) (1952) 1098–1101.
[2] N. Abramson, Information Theory and Coding, McGraw-Hill, 1963.
[3] J. Rissanen, Generalized Kraft inequality and arithmetic coding, IBM Journal of Research and Development 20 (1976) 198–203.
[4] A. Said, Introduction to arithmetic coding—theory and practice, in: K. Sayood (Ed.), Lossless Compression Handbook, Academic Press, 2003.

[5] J. Ziv, A. Lempel, A universal algorithm for data compression, IEEE Transactions on Information Theory 23 (3) (1977) 337–343.
[6] T. Welch, A technique for high-performance data compression, IEEE Computer (1984) 8–19.
[7] K. Sayood, Introduction to Data Compression, third ed., Morgan Kaufmann, 2006.
[8] K. Sayood (Ed.), Lossless Compression Handbook, Academic Press, 2003.
[9] R. Gallagher, Variations on a theme by Huffman, IEEE Transactions on Information Theory 24 (6) (1978) 668–674.
[10] J. Vitter, Design and analysis of dynamic Huffman codes, Journal of the ACM 34 (4) (1987) 825–845.
[11] ISO/IEC International Standard 10918-1, Information Technology—Digital and Coding of Continuous-Tone Still Images—Requirements and Guidelines, 1992.
[12] X. Tian, T. Le, Y. Lian, Entropy Coders of the H.264/AVD Standard: Algorithms and VLSI Architectures, Springer, 2011.
[13] I. Richardson, The H.264 Advanced Video Coding Standard, second ed., Wiley, 2010.
[14] S. Golomb, Run-length encodings, IEEE Transactions on Information Theory 12 (3) (1966) 399–401.
[15] D. Taubman, M. Marcelin, JPEG2000 Image Compression Fundamentals, Standards and Practice, Kluwer, 2002.
[16] J. Teuhola, A compression method clustered bit vectors, Information Processing Letters 7 (1978) 308–311.
[17] D. Marpe, H. Schwarz, T. Wiegand, Context-based adaptive binary arithmetic coding in the H.264/AVC video compression standard, IEEE Transactions on Circuits and Systems for Video Technology 13 (7) (2003) 620–636.
[18] C. Shannon, A mathematical theory of communication, Bell System Technical Journal (30) (1948) 623–656.
[19] F. Jelinek, Probabalistic Information Theory, McGraw Hill, 1968.
[20] J.-R. Ohm, Multimedia Communication Technology, Springer, 2003.
[21] J. Pennebaker, G. Mitchell, G. Langdon, R. Arps, An overview of the basic principles of the Q-coder adaptive binary arithmetic coder, IBM Journal of Research and Development 32 (1988) 717–726.
[22] D. Taubman, E. Ordentlich, M. Weinberger, G. Seroussic, Embedded block coding in JPEG 2000, Signal Processing: Image Communication 17 (2002) 49–72.
[23] I. Witten, R. Neal, J. Cleary, Arithmetic coding for data compression, Communications of the ACM 30 (6) (1987) 520–540.