Therefore, when coding blocks of n source symbols, the noiseless source coding theory states that for an arbitrary positive number ε, there is a variable-length code which satisfies the following:

$$H(S) \le L_{avg} < H(S) + \varepsilon \tag{5.12}$$

as n is large enough. That is, the average number of bits used in coding per source symbol is bounded below by the entropy of the source and is bounded above by the sum of the entropy and an arbitrary positive number. To make ε arbitrarily small, i.e., to make the average length of the code arbitrarily close to the entropy, we have to make the block size n large enough. This version of the noiseless coding theorem suggests a way to make the average length of a variable-length code approach the source entropy. It is known, however, that the high coding complexity that occurs when n approaches infinity makes implementation of the code impractical.

5.2 HUFFMAN CODES

Consider the source alphabet defined in Equation 5.1. The method of encoding source symbols according to their probabilities, suggested in (Shannon, 1948; Fano, 1949), is not optimum. It approaches the optimum, however, when the block size n approaches infinity. This results in a large storage requirement and high computational complexity. In many cases, we need a direct encoding method that is optimum and instantaneous (hence uniquely decodable) for an information source with finite source symbols in source alphabet S. Huffman code is the first such optimum code (Huffman, 1952), and is the technique most frequently used at present. It can be used for r-ary encoding as $r > 2$. For notational brevity, however, we discuss only the Huffman coding used in the binary case presented here.

5.2.1 REQUIRED RULES FOR OPTIMUM INSTANTANEOUS CODES

Let us rewrite Equation 5.1 as follows:

$$S = (s_1, s_2, \cdots, s_m) \tag{5.13}$$

Without loss of generality, assume the occurrence probabilities of the source symbols are as follows:

$$p(s_1) \ge p(s_2) \ge \cdots \ge p(s_{m-1}) \ge p(s_m) \tag{5.14}$$

Since we are seeking the optimum code for S, the lengths of codewords assigned to the source symbols should be

$$l_1 \le l_2 \le \cdots \le l_{m-1} \le l_m. \tag{5.15}$$

Based on the requirements of the optimum and instantaneous code, Huffman derived the following rules (restrictions):

1. $$l_1 \le l_2 \le \cdots \le l_{m-1} = l_m \tag{5.16}$$

   Equations 5.14 and 5.16 imply that when the source symbol occurrence probabilities are arranged in a nonincreasing order, the length of the corresponding codewords should be in a nondecreasing order. In other words, the codeword length of a more probable source

symbol should not be longer than that of a less probable source symbol. Furthermore, the length of the codewords assigned to the two least probable source symbols should be the same.

2. The codewords of the two least probable source symbols should be the same except for their last bits.
3. Each possible sequence of length $l_m - 1$ bits must be used either as a codeword or must have one of its prefixes used as a codeword.

Rule 1 can be justified as follows. If the first part of the rule, i.e., $l_1 \le l_2 \le \cdots \le l_{m-1}$ is violated, say, $l_1 > l_2$, then we can exchange the two codewords to shorten the average length of the code. This means the code is not optimum, which contradicts the assumption that the code is optimum. Hence it is impossible. That is, the first part of Rule 1 has to be the case. Now assume that the second part of the rule is violated, i.e., $l_{m-1} < l_m$. (Note that $l_{m-1} > l_m$ can be shown to be impossible by using the same reasoning we just used in proving the first part of the rule.) Since the code is instantaneous, codeword A_{m-1} is not a prefix of codeword A_m. This implies that the last bit in the codeword A_m is redundant. It can be removed to reduce the average length of the code, implying that the code is not optimum. This contradicts the assumption, thus proving Rule 1.

Rule 2 can be justified as follows. As in the above, A_{m-1} and A_m are the codewords of the two least probable source symbols. Assume that they do not have the identical prefix of the order $l_m - 1$. Since the code is optimum and instantaneous, codewords A_{m-1} and A_m cannot have prefixes of any order that are identical to other codewords. This implies that we can drop the last bits of A_{m-1} and A_m to achieve a lower average length. This contradicts the optimum code assumption. It proves that Rule 2 has to be the case.

Rule 3 can be justified using a similar strategy to that used above. If a possible sequence of length $l_m - 1$ has not been used as a codeword and any of its prefixes have not been used as codewords, then it can be used in place of the codeword of the mth source symbol, resulting in a reduction of the average length L_{avg}. This is a contradiction to the optimum code assumption and it justifies the rule.

5.2.2 Huffman Coding Algorithm

Based on these three rules, we see that the two least probable source symbols have codewords of equal length. These two codewords are identical except for the last bits, the binary 0 and 1, respectively. Therefore, these two source symbols can be combined to form a single new symbol. Its occurrence probability is the sum of two source symbols, i.e., $p(s_{m-1}) + p(s_m)$. Its codeword is the common prefix of order $l_m - 1$ of the two codewords assigned to s_m and s_{m-1}, respectively. The new set of source symbols thus generated is referred to as the first auxiliary source alphabet, which is one source symbol less than the original source alphabet. In the first auxiliary source alphabet, we can rearrange the source symbols according to a nonincreasing order of their occurrence probabilities. The same procedure can be applied to this newly created source alphabet. A binary 0 and a binary 1, respectively, are assigned to the last bits of the two least probable source symbols in the alphabet. The second auxiliary source alphabet will again have one source symbol less than the first auxiliary source alphabet. The procedure continues. In some step, the resultant source alphabet will have only two source symbols. At this time, we combine them to form a single source symbol with a probability of 1. The coding is then complete.

Let's go through the following example to illustrate the above Huffman algorithm.

Example 5.9

Consider a source alphabet whose six source symbols and their occurrence probabilities are listed in Table 5.9. Figure 5.1 demonstrates the Huffman coding procedure applied. In the example, among the two least probable source symbols encountered at each step we assign binary 0 to the top symbol and binary 1 to the bottom symbol.

TABLE 5.9
Source Alphabet and Huffman Codes in Example 5.9

Source Symbol	Occurrence Probability	Codeword Assigned	Length of Codeword
S_1	0.3	00	2
S_2	0.1	101	3
S_3	0.2	11	2
S_4	0.05	1001	4
S_5	0.1	1000	4
S_6	0.25	01	2

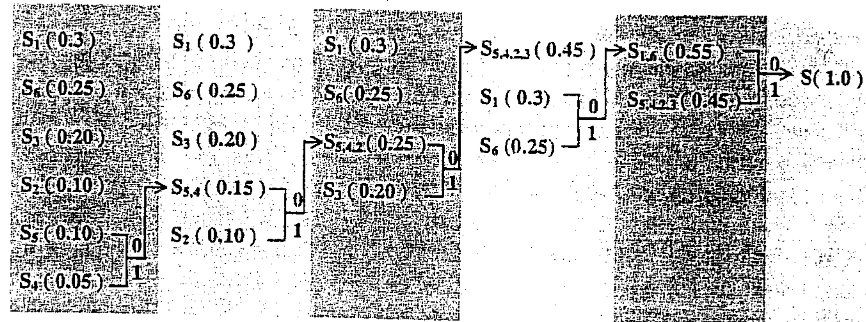

FIGURE 5.1 Huffman coding procedure in Example 5.9.

5.2.2.1 Procedures

In summary, the Huffman coding algorithm consists of the following steps.

1. Arrange all source symbols in such a way that their occurrence probabilities are in a nonincreasing order.
2. Combine the two least probable source symbols:
   - Form a new source symbol with a probability equal to the sum of the probabilities of the two least probable symbols.
   - Assign a binary 0 and a binary 1 to the two least probable symbols.
3. Repeat until the newly created auxiliary source alphabet contains only one source symbol.
4. Start from the source symbol in the last auxiliary source alphabet and trace back to each source symbol in the original source alphabet to find the corresponding codewords.

5.2.2.2 Comments

First, it is noted that the assignment of the binary 0 and 1 to the two least probable source symbols in the original source alphabet and each of the first $(u - 1)$ auxiliary source alphabets can be implemented in two different ways. Here u denotes the total number of the auxiliary source symbols in the procedure. Hence, there is a total of 2^u possible Huffman codes. In Example 5.9, there are five auxiliary source alphabets, hence a total of $2^5 = 32$ different codes. Note that each is optimum: that is, each has the same average length.

Second, in sorting the source symbols, there may be more than one symbol having equal probabilities. This results in multiple arrangements of symbols, hence multiple Huffman codes. While all of these Huffman codes are optimum, they may have some other different properties.

For instance, some Huffman codes result in the minimum codeword length variance (Sayood, 1996). This property is desired for applications in which a constant bit rate is required.

Third, Huffman coding can be applied to r-ary encoding with $r > 2$. That is, code symbols are r-ary with $r > 2$.

5.2.2.3 Applications

As a systematic procedure to encode a finite discrete memoryless source, the Huffman code has found wide application in image and video coding. Recall that it has been used in differential coding and transform coding. In transform coding, as introduced in Chapter 4, the magnitude of the quantized transform coefficients and the run-length of zeros in the zigzag scan are encoded by using the Huffman code. This has been adopted by both still image and video coding standards.

5.3 MODIFIED HUFFMAN CODES

5.3.1 MOTIVATION

As a result of Huffman coding, a set of all the codewords, called a codebook, is created. It is an agreement between the transmitter and the receiver. Consider the case where the occurrence probabilities are skewed, i.e., some are large, while some are small. Under these circumstances, the improbable source symbols take a disproportionately large amount of memory space in the codebook. The size of the codebook will be very large if the number of the improbable source symbols is large. A large codebook requires a large memory space and increases the computational complexity. A modified Huffman procedure was therefore devised in order to reduce the memory requirement while keeping almost the same optimality (Hankamer, 1979).

Example 5.10

Consider a source alphabet consisting of 16 symbols, each being a 4-bit binary sequence. That is, $S = \{s_i, i = 1,2,\cdots,16\}$. The occurrence probabilities are

$$p(s_1) = p(s_2) = 1/4,$$

$$p(s_3) = p(s_4) = \cdots = p(s_{16}) = 1/28.$$

The source entropy can be calculated as follows:

$$H(S) = 2\cdot\left(-\frac{1}{4}\log_2\frac{1}{4}\right) + 14\cdot\left(-\frac{1}{28}\log_2\frac{1}{28}\right) \approx 3.404 \quad \textit{bits per symbol}$$

Applying the Huffman coding algorithm, we find that the codeword lengths associated with the symbols are: $l_1 = l_2 = 2$, $l_3 = 4$, and $l_4 = l_5 = \cdots = l_{16} = 5$, where l_i denotes the length of the ith codeword. The average length of Huffman code is

$$L_{avg} = \sum_{i=1}^{16} p(s_i) l_i = 3.464 \quad \textit{bits per symbol}$$

We see that the average length of Huffman code is quite close to the lower entropy bound. It is noted, however, that the required codebook memory, M (defined as the sum of the codeword lengths), is quite large:

$$M = \sum_{i=1}^{16} l_i = 73 \quad bits$$

This number is obviously larger than the average codeword length multiplied by the number of codewords. This should not come as a surprise since the average here is in the statistical sense instead of in the arithmetic sense. When the total number of improbable symbols increases, the required codebook memory space will increase dramatically, resulting in a great demand on memory space.

5.3.2 Algorithm

Consider a source alphabet S that consists of 2^v binary sequences, each of length v. In other words, each source symbol is a v-bit codeword in the natural binary code. The occurrence probabilities are highly skewed and there is a large number of improbable symbols in S. The modified Huffman coding algorithm is based on the following idea: lumping all the improbable source symbols into a category named ELSE (Weaver, 1978). The algorithm is described below.

1. Categorize the source alphabet S into two disjoint groups, S_1 and S_2, such that

$$S_1 = \left\{ s_i \middle| p(s_i) > \frac{1}{2^v} \right\} \tag{5.17}$$

and

$$S_2 = \left\{ s_i \middle| p(s_i) \le \frac{1}{2^v} \right\} \tag{5.18}$$

2. Establish a source symbol ELSE with its occurrence probability equal to $p(S_2)$.
3. Apply the Huffman coding algorithm to the source alphabet S_3 with $S_3 = S_1 \cup ELSE$.
4. Convert the codebook of S_3 to that of S as follows.
   - Keep the same codewords for those symbols in S_1.
   - Use the codeword assigned to ELSE as a prefix for those symbols in S_2.

5.3.3 Codebook Memory Requirement

Codebook memory M is the sum of the codeword lengths. The M required by Huffman coding with respect to the original source alphabet S is

$$M = \sum_{i \in S} l_i = \sum_{i \in S_1} l_i + \sum_{i \in S_2} l_i \tag{5.19}$$

where l_i denotes the length of the ith codeword, as defined previously. In the case of the modified Huffman coding algorithm, the memory required M_{mH} is

$$M_{mH} = \sum_{i \in S_3} l_i = \sum_{i \in S_1} l_i + l_{ELSE} \tag{5.20}$$

where l_{ELSE} is the length of the codeword assigned to ELSE. The above equation reveals the big savings in memory requirement when the probability is skewed. The following example is used to illustrate the modified Huffman coding algorithm and the resulting dramatic memory savings.

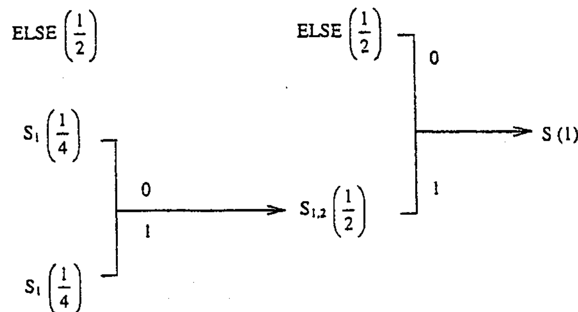

FIGURE 5.2 The modified Huffman coding procedure in Example 5.11.

Example 5.11

In this example, we apply the modified Huffman coding algorithm to the source alphabet presented in Example 5.10. We first lump the 14 symbols having the least occurrence probabilities together to form a new symbol ELSE. The probability of ELSE is the sum of the 14 probabilities. That is, $p(ELSE) = \frac{1}{28} \cdot 14 = \frac{1}{2}$.

Apply Huffman coding to the new source alphabet $S_3 = \{s_1, s_2, ELSE\}$, as shown in Figure 5.2. From Figure 5.2, it is seen that the codewords assigned to symbols s_1, s_2, and *ELSE*, respectively, are 10, 11, and 0. Hence, for every source symbol lumped into ELSE, its codeword is 0 followed by the original 4-bit binary sequence. Therefore, $M_{mH} = 2 + 2 + 1 = 5$ *bits*, i.e., the required codebook memory is only 5 bits. Compared with 73 bits required by Huffman coding (refer to Example 5.10), there is a savings of 68 bits in codebook memory space. Similar to the comment made in Example 5.10, the memory savings will be even larger if the probability distribution is skewed more severely and the number of improbable symbols is larger. The average length of the modified Huffman algorithm is $L_{avg,mH} = \frac{1}{4} \cdot 2 \cdot 2 + \frac{1}{28} \cdot 5 \cdot 14 = 3.5$ *bits per symbol*. This demonstrates that modified Huffman coding retains almost the same coding efficiency as that achieved by Huffman coding.

5.3.4 Bounds on Average Codeword Length

It has been shown that the average length of the modified Huffman codes satisfies the following condition:

$$H(S) \le L_{avg} < H(S) + 1 - p \log_2 p \tag{5.21}$$

where $p = \sum_{s_i \in s_2} p(s_i)$. It is seen that, compared with the noiseless source coding theorem, the upper bound of the code average length is increased by a quantity of $-p \log_2 p$. In Example 5.11 it is seen that the average length of the modified Huffman code is close to that achieved by the Huffman code. Hence the modified Huffman code is almost optimum.

5.4 ARITHMETIC CODES

Arithmetic coding, which is quite different from Huffman coding, is gaining increasing popularity. In this section, we will first analyze the limitations of Huffman coding. Then the principle of arithmetic coding will be introduced. Finally some implementation issues are discussed briefly.