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

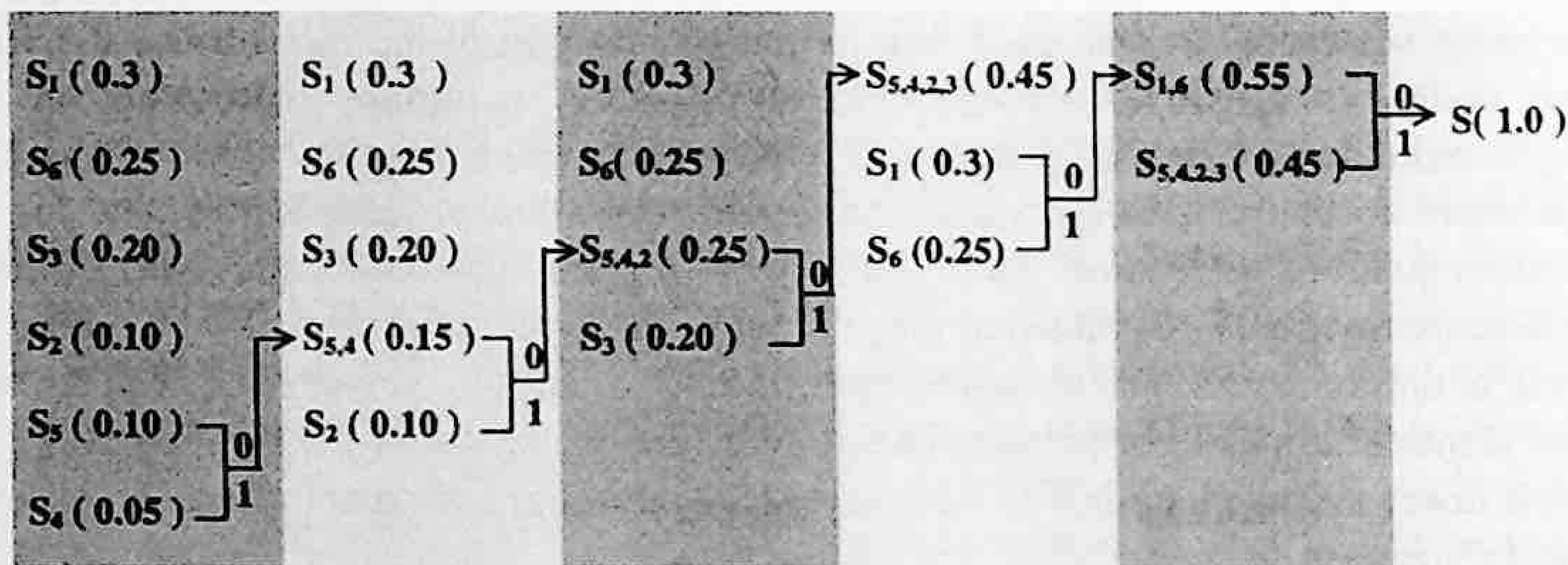


FIGURE 5.1 Huffman coding procedure in Example 5.9.

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:

IPR2021-00827
Unified EX1008 Page 143

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

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.

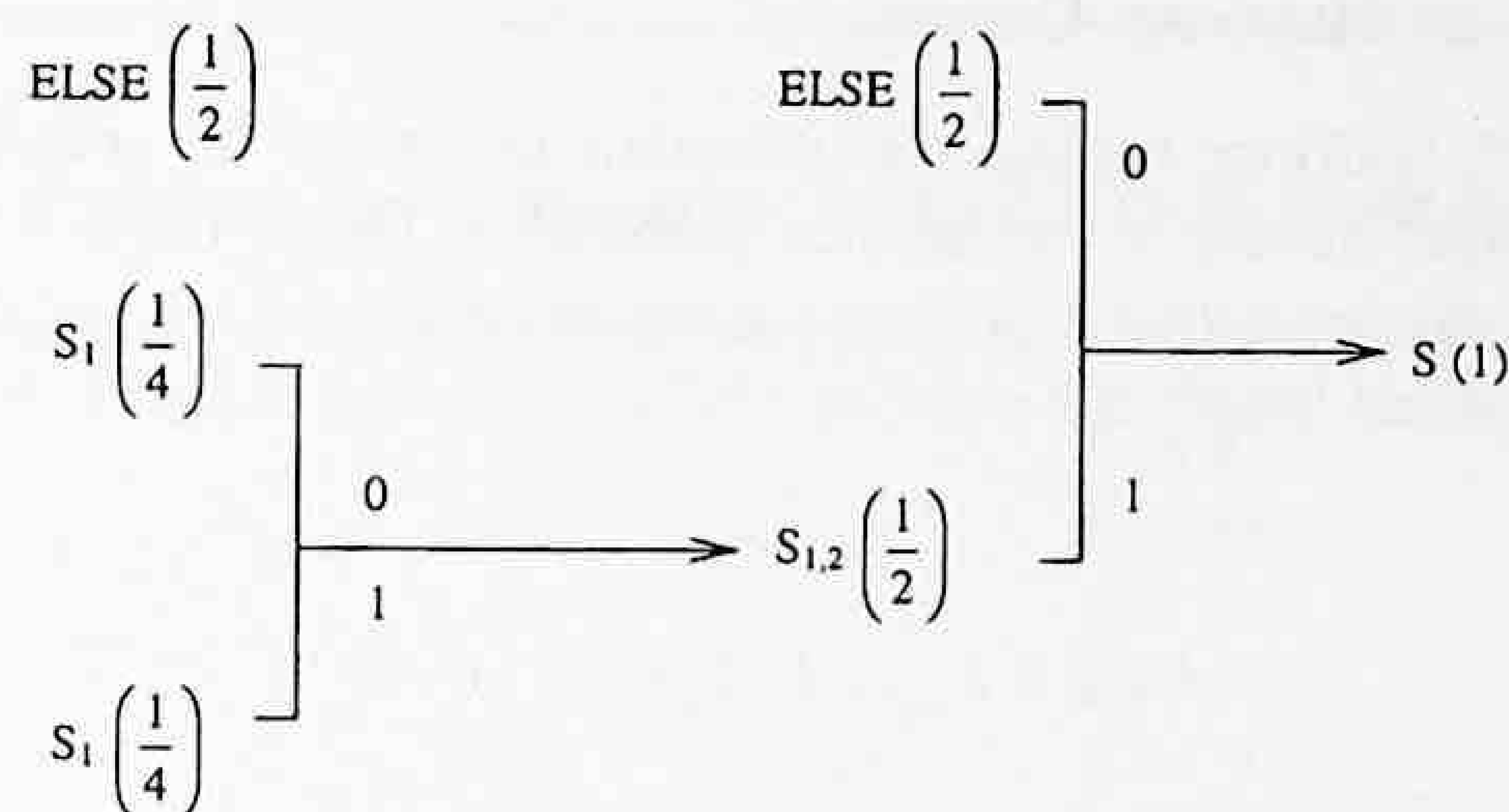


FIGURE 5.2 The modified Huffman coding procedure in Example 5.11.

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 $p = \Sigma_{s_i \in s_2} p(s_i)$. 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.

IPR2021-00827
Unified EX1008 Page 145

5.4.1 Limitations of Huffman Coding

As seen in Section 5.2, Huffman coding is a systematic procedure for encoding a source alphabet, with each source symbol having an occurrence probability. Under these circumstances, Huffman coding is optimum in the sense that it produces a minimum coding redundancy. It has been shown that the average codeword length achieved by Huffman coding satisfies the following inequality (Gallagher, 1978).

$$H(S) \le L_{avg} < H(S) + p_{\max} + 0.086 \tag{5.22}$$

where $H(S)$ is the entropy of the source alphabet, and $p_{\max}$ denotes the maximum occurrence probability in the set of the source symbols. This inequality implies that the upper bound of the average codeword length of Huffman code is determined by the entropy and the maximum occurrence probability of the source symbols being encoded.

In the case where the probability distribution among source symbols is skewed (some probabilities are small, while some are quite large), the upper bound may be large, implying that the coding redundancy may not be small. Imagine the following extreme situation. There are only two source symbols. One has a very small probability, while the other has a very large probability (very close to 1). The entropy of the source alphabet in this case is close to 0 since the uncertainty is very small. Using Huffman coding, however, we need two bits: one for each. That is, the average codeword length is 1, which means that the redundancy is very close to 1. This agrees with Equation 5.22. This inefficiency is due to the fact that Huffman coding always encodes a source symbol with an integer number of bits.

The noiseless coding theorem (reviewed in Section 5.1) indicates that the average codeword length of a block code can approach the source alphabet entropy when the block size approaches infinity. As the block size approaches infinity, the storage required, the codebook size, and the coding delay will approach infinity, however, and the complexity of the coding will be out of control.

The fundamental idea behind Huffman coding and Shannon-Fano coding (devised a little earlier than Huffman coding [Bell et al., 1990]) is block coding. That is, some codeword having an integral number of bits is assigned to a source symbol. A message may be encoded by cascading the relevant codewords. It is the *block-based* approach that is responsible for the limitations of Huffman codes.

Another limitation is that when encoding a message that consists of a sequence of source symbols, the nth extension Huffman coding needs to enumerate all possible sequences of source symbols having the same length, as discussed in coding the nth extended source alphabet. This is not computationally efficient.

Quite different from Huffman coding, arithmetic coding is *stream-based*. It overcomes the drawbacks of Huffman coding. A string of source symbols is encoded as a string of code symbols. Hence, it is free of the integral-bits-per-source symbol restriction and is more efficient. Arithmetic coding may reach the theoretical bound to coding efficiency specified in the noiseless source coding theorem for any information source. Below, we introduce the principle of arithmetic coding, from which we can see the stream-based nature of arithmetic coding.

5.4.2 Principle of Arithmetic Coding

To understand the different natures of Huffman coding and arithmetic coding, let us look at Example 5.12, where we use the same source alphabet and the associated occurrence probabilities used in Example 5.9. In this example, however, a string of source symbols $s_1 s_2 s_3 s_4 s_5 s_6$ is encoded. Note that we consider the terms *string* and *stream* to be slightly different. By stream, we mean a message, or possibly several messages, which may correspond to quite a long sequence of source symbols. Moreover, stream gives a dynamic "flavor." Later on we will see that arithmetic coding

IPR2021-00827
Unified EX1008 Page 146

TABLE 5.10
Source Alphabet and Cumulative Probabilities in Example 5.12

Source Symbol	Occurrence Probability	Associated Subintervals	CP
S_1	0.3	[0, 0.3)	0
S_2	0.1	[0.3, 0.4)	0.3
S_3	0.2	[0.4, 0.6)	0.4
S_4	0.05	[0.6, 0.65)	0.6
S_5	0.1	[0.65, 0.75)	0.65
S_6	0.25	[0.75, 1.0)	0.75

is implemented in an incremental manner. Hence stream is a suitable term to use for arithmetic coding. In this example, however, only six source symbols are involved. Hence we consider the term *string* to be suitable, aiming at distinguishing it from the term *block*.

Example 5.12
The set of six source symbols and their occurrence probabilities are listed in Table 5.10. In this example, the string to be encoded using arithmetic coding is $s_1s_2s_3s_4s_5s_6$. In the following four subsections we will use this example to illustrate the principle of arithmetic coding and decoding.

5.4.2.1 Dividing Interval [0,1) into Subintervals

As pointed out by Elias, it is not necessary to sort out source symbols according to their occurrence probabilities. Therefore in Figure 5.3(a) the six symbols are arranged in their natural order, from symbols s_1, s_2, $\cdots$, up to s_6. The real interval between 0 and 1 is divided into six subintervals, each having a length of $p(s_i)$, $i = 1,2,\cdots,6$. Specifically, the interval denoted by [0,1) — where 0 is included in (the left end is closed) and 1 is excluded from (the right end is open) the interval — is divided into six subintervals. The first subinterval [0, 0.3) corresponds to s_1 and has a length of $P(s_1)$, i.e., 0.3. Similarly, the subinterval [0, 0.3) is said to be closed on the left and open on the right. The remaining five subintervals are similarly constructed. All six subintervals thus formed are disjoint and their union is equal to the interval [0, 1). This is because the sum of all the probabilities is equal to 1.

We list the sum of the preceding probabilities, known as *cumulative probability* (Langdon, 1984), in the right-most column of Table 5.10 as well. Note that the concept of cumulative probability (CP) is slightly different from that of cumulative distribution function (CDF) in probability theory. Recall that in the case of discrete random variables the CDF is defined as follows.

$$CDF(s_i) = \sum_{j=1}^{i} p(s_j) \tag{5.23}$$

The CP is defined as

$$CP(s_i) = \sum_{j=1}^{i-1} p(s_j) \tag{5.24}$$

where $CP(s_1) = 0$ is defined. Now we see each subinterval has its lower end point located at $CP(s_i)$. The width of each subinterval is equal to the probability of the corresponding source symbol. A subinterval can be completely defined by its lower end point and its width. Alternatively, it is

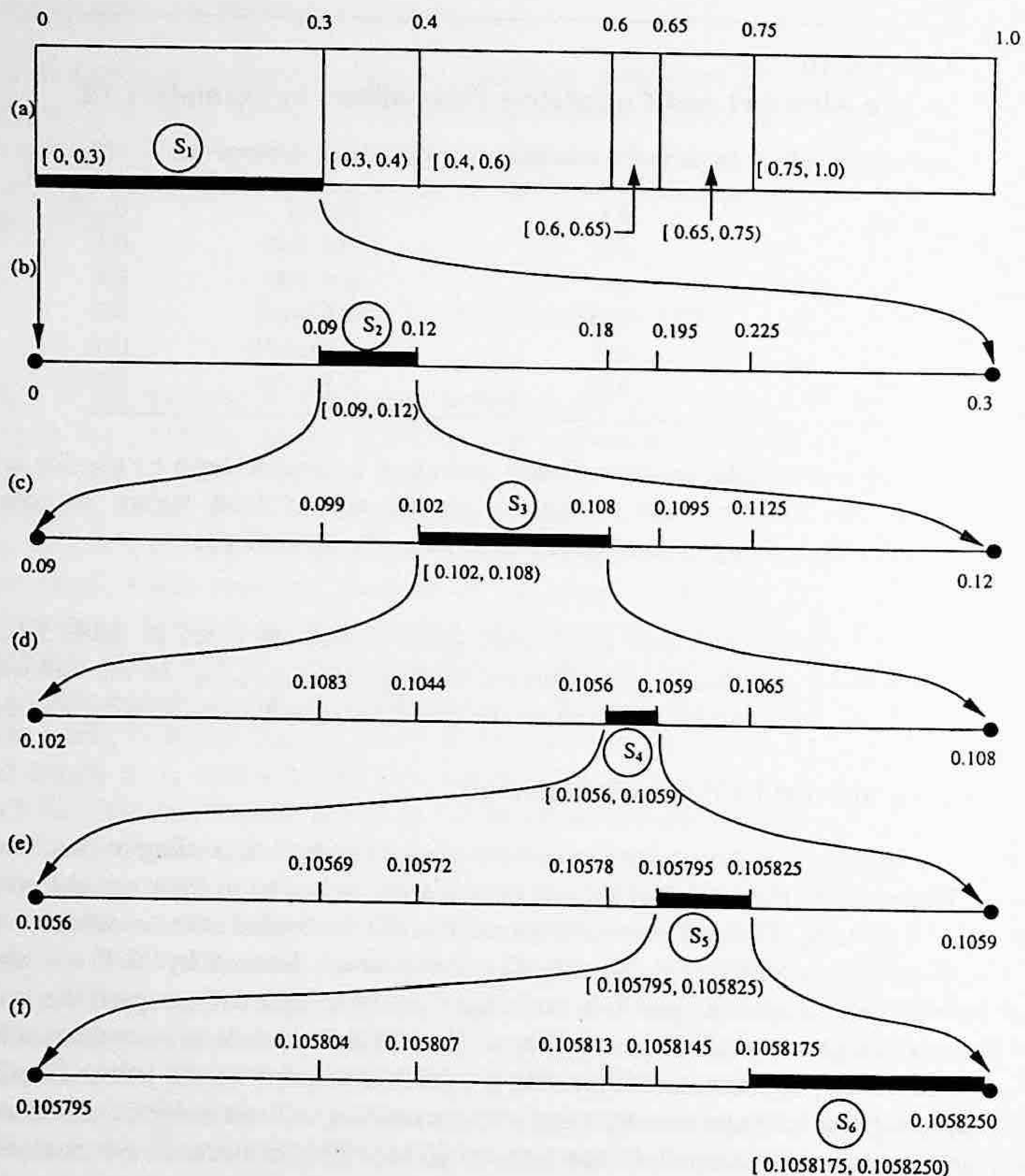


FIGURE 5.3 Arithmetic coding working on the same source alphabet as that in Example 5.9. The encoded symbol string is S_1 S_2 S_3 S_4 S_5 S_6.

determined by its two end points: the lower and upper end points (sometimes also called the left and right end points).

Now we consider encoding the string of source symbols $s_1s_2s_3s_4s_5s_6$ with the arithmetic coding method.

5.4.2.2 Encoding

Encoding the First Source Symbol

Refer to Figure 5.3(a). Since the first symbol is s_1, we pick up its subinterval [0, 0.3). Picking up the subinterval [0, 0.3) means that any real number in the subinterval, i.e., any real number equal to or greater than 0 and smaller than 0.3, can be a pointer to the subinterval, thus representing the source symbol s_1. This can be justified by considering that all the six subintervals are disjoint.

Encoding the Second Source Symbol

Refer to Figure 5.3(b). We use the same procedure as used in part (a) to divide the interval [0, 0.3) into six subintervals. Since the second symbol to be encoded is s_2, we pick up its subinterval [0.09, 0.12).

Notice that the subintervals are recursively generated from part (a) to part (b). It is known that an interval may be completely specified by its lower end point and width. Hence, the subinterval recursion in the arithmetic coding procedure is equivalent to the following two recursions: end point recursion and width recursion.

From interval [0, 0.3) derived in part (a) to interval [0.09, 0.12) obtained in part (b), we can conclude the following lower end point recursion:

$$L_{new} = L_{current} + W_{current} \cdot CP_{new} \tag{5.25}$$

where L_{new}, $L_{current}$ represent, respectively, the lower end points of the new and current recursions, and the $W_{current}$ and the CP_{new} denote, respectively, the width of the interval in the current recursion and the cumulative probability in the new recursion. The width recursion is

$$W_{new} = W_{current} \cdot p(s_i) \tag{5.26}$$

where W_{new}, and $p(s_i)$ are, respectively, the width of the new subinterval and the probability of the source symbol s_i that is being encoded. These two recursions, also called double recursion (Langdon, 1984), play a central role in arithmetic coding.

Encoding the Third Source Symbol

Refer to Figure 5.3(c). When the third source symbol is encoded, the subinterval generated above in part (b) is similarly divided into six subintervals. Since the third symbol to encode is s_3, its subinterval [0.102, 0.108) is picked up.

Encoding the Fourth, Fifth, and Sixth Source Symbols

Refer to Figure 5.3(d,e,f). The subinterval division is carried out according to Equations 5.25 and 5.26. The symbols s_4, s_5, and s_6 are encoded. The final subinterval generated is [0.1058175, 0.1058250).

That is, the resulting subinterval [0.1058175, 0.1058250) can represent the source symbol string $s_1 s_2 s_3 s_4 s_5 s_6$. Note that in this example decimal digits instead of binary digits are used. In binary arithmetic coding, the binary digits 0 and 1 are used.

5.4.2.3 Decoding

As seen in this example, for the encoder of arithmetic coding, the input is a source symbol string and the output is a subinterval. Let us call this the final subinterval or the resultant subinterval. Theoretically, any real numbers in the interval can be the code string for the input symbol string since all subintervals are disjoint. Often, however, the lower end of the final subinterval is used as the code string. Now let us examine how the decoding process is carried out with the lower end of the final subinterval.

Decoding sort of reverses what encoding has done. The decoder knows the encoding procedure and therefore has the information contained in Figure 5.3(a). It compares the lower end point of the final subinterval 0.1058175 with all the end points in (a). It is determined that $0 < 0.1058175 < 0.3$. That is, the lower end falls into the subinterval associated with the symbol s_1. Therefore, the symbol s_1 is first decoded.

Once the first symbol is decoded, the decoder may know the partition of subintervals shown in Figure 5.3(b). It is then determined that $0.09 < 0.1058175 < 0.12$. That is, the lower end is contained in the subinterval corresponding to the symbol s_2. As a result, s_2 is the second decoded symbol.

The procedure repeats itself until all six symbols are decoded. That is, based on Figure 5.3(c), it is found that $0.102 < 0.1058175 < 0.108$. The symbol s_3 is decoded. Then, the symbols s_4, s_5, s_6 are subsequently decoded because the following inequalities are determined:

$$0.1056 < 0.1058175 < 0.1059$$

$$0.105795 < 0.1058175 < 0.1058250$$

$$0.1058145 < 0.1058175 < 0.1058250$$

Note that a terminal symbol is necessary to inform the decoder to stop decoding.

The above procedure gives us an idea of how decoding works. The decoding process, however, does not need to construct parts (b), (c), (d), (e), and (f) of Figure 5.3. Instead, the decoder only needs the information contained in Figure 5.3(a). Decoding can be split into the following three steps: *comparison*, *readjustment* (subtraction), and *scaling* (Langdon, 1984).

As described above, through comparison we decode the first symbol s_1. From the way Figure 5.3(b) is constructed, we know the decoding of s_2 can be accomplished as follows. We subtract the lower end of the subinterval associated with s_1 in part (a), that is, 0 in this example, from the lower end of the final subinterval 0.1058175, resulting in 0.1058175. Then we divide this number by the width of the subinterval associated with s_1, i.e., the probability of s_1, 0.3, resulting in 0.352725. Looking at part (a) of Figure 5.3, it is found that $0.3 < 0.352725 < 0.4$. That is, the number is within the subinterval corresponding to s_2. Therefore the second decoded symbol is s_2. Note that these three decoding steps exactly "undo" what encoding did.

To decode the third symbol, we subtract the lower end of the subinterval with s_2, 0.3 from 0.352725, obtaining 0.052725. This number is divided by the probability of s_2, 0.1, resulting in 0.52725. The comparison of 0.52725 with end points in part (a) reveals that the third decoded symbol is s_3.

In decoding the fourth symbol, we first subtract the lower end of the s_3's subinterval in part (a), 0.4 from 0.52725, getting 0.12725. Dividing 0.12725 by the probability of s_3, 0.2, results in 0.63625. Referring to part (a), we decode the fourth symbol as s_4 by comparison.

Subtraction of the lower end of the subinterval of s_4 in part (a), 0.6, from 0.63625 leads to 0.03625. Division of 0.03625 by the probability of s_4, 0.05, produces 0.725. The comparison between 0.725 and the end points in part (a) decodes the fifth symbol as s_5.

Subtracting 0.725 by the lower end of the subinterval associated with s_5 in part (a), 0.65, gives 0.075. Dividing 0.075 by the probability of s_5, 0.1, generates 0.75. The comparison indicates that the sixth decoded symbol is s_6.

In summary, considering the way in which parts (b), (c), (d), (e), and (f) of Figure 5.3 are constructed, we see that the three steps discussed in the decoding process: comparison, readjustment, and scaling, exactly "undo" what the encoding procedure has done.

5.4.2.4 Observations

Both encoding and decoding involve only arithmetic operations (addition and multiplication in encoding, subtraction and division in decoding). This explains the name *arithmetic coding*.

We see that an input source symbol string $s_1 s_2 s_3 s_4 s_5 s_6$, via encoding, corresponds to a subinterval [0.1058175, 0.1058250). Any number in this interval can be used to denote the string of the source symbols.

We also observe that arithmetic coding can be carried out in an *incremental* manner. That is, source symbols are fed into the encoder one by one and the final subinterval is refined continually, i.e., the code string is generated continually. Furthermore, it is done in a manner called *first in first out* (FIFO). That is, the source symbol encoded first is decoded first. This manner is superior to that of *last in first out* (LIFO). This is because FIFO is suitable for adaptation to the statistics of the symbol string.

It is obvious that the width of the final subinterval becomes smaller and smaller when the length of the source symbol string becomes larger and larger. This causes what is known as the precision

IPR2021-00827
Unified EX1008 Page 150

problem. It is this problem that prohibited arithmetic coding from practical usage for quite a long period of time. Only after this problem was solved in the late 1970s, did arithmetic coding become an increasingly important coding technique.

It is necessary to have a termination symbol at the end of an input source symbol string. In this way, an arithmetic coding system is able to know when to terminate decoding.

Compared with Huffman coding, arithmetic coding is quite different. Basically, Huffman coding converts each source symbol into a fixed codeword with an integral number of bits, while arithmetic coding converts a source symbol string to a code symbol string. To encode the same source symbol string, Huffman coding can be implemented in two different ways. One way is shown in Example 5.9. We construct a fixed codeword for each source symbol. Since Huffman coding is instantaneous, we can cascade the corresponding codewords to form the output, a 17-bit code string 00.101.11.1001.1000.01, where, for easy reading, the five periods are used to indicate different codewords. As we see that for the same source symbol string, the final subinterval obtained by using arithmetic coding is [0.1058175, 0.1058250). It is noted that a decimal in binary number system, 0.000110111111111, which is of 15 bits, is equal to the decimal in decimal number system, 0.1058211962, which falls into the final subinterval representing the string $s_1s_2s_3s_4s_5s_6$. This indicates that, for this example, arithmetic coding is more efficient than Huffamn coding.

Another way is to form a sixth extension of the source alphabet as discussed in Section 5.1.4: treat each group of six source symbols as a new source symbol; calculate its occurrence probability by multiplying the related six probabilities; then apply the Huffman coding algorithm to the sixth extension of the discrete memoryless source. This is called the sixth extension of Huffman block code (refer to Section 5.1.2.2). In other words, in order to encode the source string $s_1s_2s_3s_4s_5s_6$, Huffman coding encodes all of the $6^6 = 46{,}656$ codewords in the sixth extension of the source alphabet. This implies a high complexity in implementation and a large codebook. It is therefore not efficient.

Note that we use the decimal fraction in this section. In binary arithmetic coding, we use the binary fraction. In (Langdon, 1984) both binary source and code alphabets are used in binary arithmetic coding.

Similar to the case of Huffman coding, arithmetic coding is also applicable to r-ary encoding with $r > 2$.

5.4.3 Implementation Issues

As mentioned, the final subinterval resulting from arithmetic encoding of a source symbol string becomes smaller and smaller as the length of the source symbol string increases. That is, the lower and upper bounds of the final subinterval become closer and closer. This causes a growing precision problem. It is this problem that prohibited arithmetic coding from practical usage for a long period of time. This problem has been resolved and the finite precision arithmetic is now used in arithmetic coding. This advance is due to the incremental implementation of arithmetic coding.

5.4.3.1 Incremental Implementation

Recall Example 5.12. As source symbols come in one by one, the lower and upper ends of the final subinterval get closer and closer. In Figure 5.3, these lower and upper ends in Example 5.12 are listed. We observe that after the third symbol, s_3, is encoded, the resultant subinterval is [0.102, 0.108). That is, the two most significant decimal digits are the same and they remain the same in the encoding process. Hence, we can transmit these two digits without affecting the final code string. After the fourth symbol s_4 is encoded, the resultant subinterval is [0.1056, 0.1059). That is, one more digit, 5, can be transmitted. Or we say the cumulative output is now .105. After the sixth symbol is encoded, the final subinterval is [0.1058175, 0.1058250). The cumulative output is 0.1058. Refer to Table 5.11. This important observation reveals that we are able to incrementally transmit output (the code symbols) and receive input (the source symbols that need to be encoded).

TABLE 5.11
Final Subintervals and Cumulative Output in Example 5.12

Source Symbol	Final Subinterval		Cumulative Output
	Lower End	Upper End	
S_1	0	0.3	—
S_2	0.09	0.12	—
S_3	0.102	0.108	0.10
S_4	0.1056	0.1059	0.105
S_5	0.105795	0.105825	0.105
S_6	0.1058175	0.1058250	0.1058

5.4.3.2 Finite Precision

With the incremental manner of transmission of encoded digits and reception of input source symbols, it is possible to use finite precision to represent the lower and upper bounds of the resultant subinterval, which gets closer and closer as the length of the source symbol string becomes long.

Instead of floating-point math, integer math is used. The potential problems known as underflow and overflow, however, need to be carefully monitored and controlled (Bell et al., 1990).

5.4.3.3 Other Issues

There are some other problems that need to be handled in implementation of binary arithmetic coding. Two of them are listed below (Langdon and Rissanen, 1981).

Eliminating Multiplication
The multiplication in the recursive division of subintervals is expensive in hardware as well as software. It can be avoided in binary arithmetic coding so as to simplify the implementation of binary arithmetic coding. The idea is to approximate the lower end of the interval by the closest binary fraction 2^{-Q}, where Q is an integer. Consequently, the multiplication by 2^{-Q} becomes a right shift by Q bits. A simpler approximation to eliminate multiplication is used in the Skew Coder (Langdon and Rissanen, 1982) and the Q-Coder (Pennebaker et al., 1988).

Carry-Over Problem
Carry-over takes place in the addition required in the recursion updating the lower end of the resultant subintervals. A carry may *propagate* over q bits. If the q is larger than the number of bits in the fixed-length register utilized in finite precision arithmetic, the carry-over problem occurs. To block the carry-over problem, a technique known as "bit stuffing" is used, in which an additional buffer register is utilized.

For a detailed discussion on the various issues involved, readers are referred to (Langdon et al., 1981, 1982, 1984; Pennebaker et al., 1988, 1992). Some computer programs of arithmetic coding in C language can be found in (Bell et al., 1990; Nelson and Gailley, 1996).

5.4.4 History

The idea of encoding by using cumulative probability in some ordering, and decoding by comparison of magnitude of binary fraction, was introduced in Shannon's celebrated paper (Shannon, 1948). The recursive implementation of arithmetic coding was devised by Elias. This unpublished result was first introduced by Abramson as a note in his book on information theory and coding

(Abramson, 1963). The result was further developed by Jelinek in his book on information theory (Jelinek, 1968). The growing precision problem prevented arithmetic coding from attaining practical usage, however. The proposal of using finite precision arithmetic was made independently by Pasco (Pasco, 1976) and Rissanen (Rissanen, 1976). Practical arithmetic coding was developed by several independent groups (Rissanen and Langdon, 1979; Rubin, 1979; Guazzo, 1980). A well-known tutorial paper on arithmetic coding appeared in (Langdon, 1984). The tremendous efforts made in IBM led to a new form of adaptive binary arithmetic coding known as the Q-coder (Pennebaker et al., 1988). Based on the Q-coder, the activities of the international still image coding standards JPEG and JBIG combined the best features of the various existing arithmetic coders and developed the binary arithmetic coding procedure known as the QM-coder (Pennebaker and Mitchell, 1992).

5.4.5 Applications

Arithmetic coding is becoming popular. Note that in text and bilevel image applications there are only two source symbols (black and white), and the occurrence probability is skewed. Therefore binary arithmetic coding achieves high coding efficiency. It has been successfully applied to bilevel image coding (Langdon and Rissanen, 1981) and adopted by the international standards for bilevel image compression, JBIG. It has also been adopted by the international still image coding standard, JPEG. More in this regard is covered in the next chapter when we introduce JBIG.

5.5 SUMMARY

So far in this chapter, not much has been explicitly discussed regarding the term variable-length codes. It is known that if source symbols in a source alphabet are equally probable, i.e., their occurrence probabilities are the same, then fixed-length codes such as the natural binary code are a reasonable choice. When the occurrence probabilities, however, are unequal, variable-length codes should be used in order to achieve high coding efficiency. This is one of the restrictions on the minimum redundancy codes imposed by Huffman. That is, the length of the codeword assigned to a probable source symbol should not be larger than that associated with a less probable source symbol. If the occurrence probabilities happen to be the integral powers of 1/2, then choosing the codeword length equal to $-\log_2 p(s_i)$ for a source symbol s_i having the occurrence probability $p(s_i)$ results in minimum redundancy coding. In fact, the average length of the code thus generated is equal to the source entropy.

Huffman devised a systematic procedure to encode a source alphabet consisting of finitely many source symbols, each having an occurrence probability. It is based on some restrictions imposed on the optimum, instantaneous codes. By assigning codewords with variable lengths according to variable probabilities of source symbols, Huffman coding results in minimum redundancy codes, or optimum codes for short. These have found wide applications in image and video coding and have been adopted in the international still image coding standard JPEG and video coding standards H.261, H.263, and MPEG 1 and 2.

When some source symbols have small probabilities and their number is large, the size of the codebook of Huffman codes will require a large memory space. The modified Huffman coding technique employs a special symbol to lump all the symbols with small probabilities together. As a result, it can reduce the codebook memory space drastically while retaining almost the same coding efficiency as that achieved by the conventional Huffman coding technique.

On the one hand, Huffman coding is optimum as a block code for a fixed-source alphabet. On the other hand, compared with the source entropy (the lower bound of the average codeword length) it is not efficient when the probabilities of a source alphabet are skewed with the maximum probability being large. This is caused by the restriction that Huffman coding can only assign an integral number of bits to each codeword.

Another limitation of Huffman coding is that it has to enumerate and encode all the possible groups of n source symbols in the nth extension Huffman code, even though there may be only one such group that needs to be encoded.

Arithmetic coding can overcome the limitations of Huffman coding because it is stream-oriented rather than block-oriented. It translates a stream of source symbols into a stream of code symbols. It can work in an incremental manner. That is, the source symbols are fed into the coding system one by one and the code symbols are output continually. In this stream-oriented way, arithmetic coding is more efficient. It can approach the lower coding bounds set by the noiseless source coding theorem for various sources.

The recursive subinterval division (equivalently, the double recursion: the lower end recursion and width recursion) is the heart of arithmetic coding. Several measures have been taken in the implementation of arithmetic coding. They include the incremental manner, finite precision, and the elimination of multiplication. Due to its merits, binary arithmetic coding has been adopted by the international bilevel image coding standard, JBIG, and still image coding standard, JPEG. It is becoming an increasingly important coding technique.

5.6 EXERCISES

5-1. What does the noiseless source coding theorem state (using your own words)? Under what condition does the average code length approach the source entropy? Comment on the method suggested by the noiseless source coding theorem.

5-2. What characterizes a block code? Consider another definition of block code in (Blahut, 1986): a block code breaks the input data stream into blocks of fixed length n and encodes each block into a codeword of fixed length m. Are these two definitions (the one above and the one in Section 5.1, which comes from [Abramson, 1963]) essentially the same? Explain.

5-3. Is a uniquely decodable code necessarily a prefix condition code?

5-4. For text encoding, there are only two source symbols for black and white. It is said that Huffman coding is not efficient in this application. But it is known as the optimum code. Is there a contradiction? Explain.

5-5. A set of source symbols and their occurrence probabilities is listed in Table 5.12. Apply the Huffman coding algorithm to encode the alphabet.

5-6. Find the Huffman code for the source alphabet shown in Example 5.10.

5-7. Consider a source alphabet $S = \{s_i, i = 1,2,\cdots,32\}$ with $p(s_1) = 1/4$, $p(s_i) = 3/124$, $i = 2,3,\cdots,32$. Determine the source entropy and the average length of Huffman code if applied to the source alphabet. Then apply the modified Huffman coding algorithm. Calculate the average length of the modified Huffman code. Compare the codebook memory required by Huffman code and the modified Huffman code.

5-8. A source alphabet consists of the following four source symbols: s_1, s_2, s_3, and s_4, with their occurrence probabilities equal to 0.25, 0.375, 0.125, and 0.25, respectively. Applying arithmetic coding as shown in Example 5.12 to the source symbol string $s_2s_1s_3s_4$, determine the lower end of the final subinterval.

5-9. For the above problem, show step by step how we can decode the original source string from the lower end of the final subinterval.

5-10. In Problem 5.8, find the codeword of the symbol string $s_2s_1s_3s_4$ by using the fourth extension of the Huffman code. Compare the two methods.

5-11. Discuss how modern arithmetic coding overcame the growing precision problem.

IPR2021-00827
Unified EX1008 Page 154

TABLE 5.12
Source Alphabet in Problem 5.5

Source Symbol	Occurrence Probability	Codeword Assigned
S_1	0.20	
S_2	0.18	
S_3	0.10	
S_4	0.10	
S_5	0.10	
S_6	0.06	
S_7	0.06	
S_8	0.04	
S_9	0.04	
S_{10}	0.04	
S_{11}	0.04	
S_{12}	0.04	

REFERENCES

Abramson, N. *Information Theory and Coding*, New York: McGraw-Hill, 1963.

Bell, T. C., J. G. Cleary, and I. H. Witten, *Text Compression*, Englewood, NJ: Prentice-Hall, 1990.

Blahut, R. E. *Principles and Practice of Information Theory*, Reading, MA: Addison-Wesley, 1986.

Fano, R. M. The transmission of information, Tech. Rep. 65, Research Laboratory of Electronics, MIT, Cambridge, MA, 1949.

Gallagher, R. G. Variations on a theme by Huffman, *IEEE Trans. Inf. Theory,* IT-24(6), 668-674, 1978.

Guazzo, M. A general minimum-redundacy source-coding algorithm, *IEEE Trans. Inf. Theory,* IT-26(1), 15-25, 1980.

Hankamer, M. A modified Huffman procedure with reduced memory requirement, *IEEE Trans. Commun.,* COM-27(6), 930-932, 1979.

Huffman, D. A. A method for the construction of minimum-redundancy codes, *Proc. IRE,* 40, 1098-1101, 1952.

Jelinek, F. *Probabilistic Information Theory*, New York: McGraw-Hill, 1968.

Langdon, G. G., Jr. and J. Rissanen, Compression of black-white images with arithmetic coding, *IEEE Trans. Commun.,* COM-29(6), 858-867, 1981.

Langdon, G. G., Jr. and J. Rissanen, A simple general binary source code, *IEEE Trans. Inf. Theory,* IT-28, 800, 1982.

Langdon, G. G., Jr., An introduction to arithmetic coding, *IBM J. Res. Dev.,* 28(2), 135-149, 1984.

Nelson, M. and J. Gailly, *The Data Compression Book*, 2nd ed., New York: M&T Books, 1996.

Pasco, R. Source Coding Algorithms for Fast Data Compression, Ph.D. dissertation, Stanford University, Stanford, CA, 1976.

Pennebaker, W. B., J. L. Mitchell, G. G. Langdon, Jr., and R. B. Arps, An overview of the basic principles of the Q-coder adaptive binary arithmetic Coder, *IBM J. Res. Dev.,* 32(6), 717-726, 1988.

Pennebaker, W. B. and J. L. Mitchell, *JPEG: Still Image Data Compression Standard*, New York: Van Nostrand Reinhold, 1992.

Rissanen, J. J. Generalized Kraft inequality and arithmetic coding, *IBM J. Res. Dev.,* 20, 198-203, 1976.

Rissanen, J. J. and G. G. Landon, Arithmetic coding, *IBM J. Res. Dev.,* 23(2), 149-162, 1979.

Rubin, F. Arithmetic stream coding using fixed precision registers, *IEEE Trans. Inf. Theory,* IT-25(6), 672-675, 1979.

Sayood, K. *Introduction to Data Compression*, San Francisco, CA: Morgan Kaufmann Publishers, 1996.

Shannon, C. E. A mathematical theory of communication, *Bell Syst. Tech. J.,* 27, 379-423, 1948; 623-656, 1948.

Weaver, C. S., Digital ECG data compression, in *Digital Encoding of Electrocardiograms,* H. K. Wolf, Ed., Springer-Verlag, Berlin/New York, 1979.

IPR2021-00827
Unified EX1008 Page 156

6 Run-Length and Dictionary Coding: Information Theory Results (III)

As mentioned at the beginning of Chapter 5, we are studying some codeword assignment (encoding) techniques in Chapters 5 and 6. In this chapter, we focus on run-length and dictionary-based coding techniques. We first introduce Markov models as a type of dependent source model in contrast to the memoryless source model discussed in Chapter 5. Based on the Markov model, run-length coding is suitable for facsimile encoding. Its principle and application to facsimile encoding are discussed, followed by an introduction to dictionary-based coding, which is quite different from Huffman and arithmetic coding techniques covered in Chapter 5. Two types of adaptive dictionary coding techniques, the LZ77 and LZ78 algorithms, are presented. Finally, a brief summary of and a performance comparison between international standard algorithms for lossless still image coding are presented.

Since the Markov source model, run-length, and dictionary-based coding are the core of this chapter, we consider this chapter as a third part of the information theory results presented in the book. It is noted, however, that the emphasis is placed on their applications to image and video compression.

6.1 MARKOV SOURCE MODEL

In the previous chapter we discussed the discrete memoryless source model, in which source symbols are assumed to be independent of each other. In other words, the source has zero memory, i.e., the previous status does not affect the present one at all. In reality, however, many sources are dependent in nature. Namely, the source has memory in the sense that the previous status has an influence on the present status. For instance, as mentioned in Chapter 1, there is an interpixel correlation in digital images. That is, pixels in a digital image are not independent of each other. As will be seen in this chapter, there is some dependence between characters in text. For instance, the letter u often follows the letter q in English. Therefore it is necessary to introduce models that can reflect this type of dependence. A Markov source model is often used in this regard.

6.1.1 DISCRETE MARKOV SOURCE

Here, as in the previous chapter, we denote a source alphabet by $S = \{s_1, s_2, \cdots, s_m\}$ and the occurrence probability by p. An lth order Markov source is characterized by the following equation of conditional probabilities.

$$p\left(s_j \middle| s_{i1}, s_{i2}, \cdots, s_{il}, \cdots\right) = p\left(s_j \middle| s_{i1}, s_{i2}, \cdots, s_{il}\right), \quad (6.1)$$

where $j, i1, i2, \cdots, il, \cdots \in \{1,2,\cdots,m\}$, i.e., the symbols $s_j, s_{i1}, s_{i2}, \cdots, s_{il}, \cdots$ are chosen from the source alphabet S. This equation states that the source symbols are not independent of each other. The occurrence probability of a source symbol is determined by some of its previous symbols. Specifically, the probability of s_j given its history being $s_{i1}, s_{i2}, \cdots, s_{il}, \cdots$ (also called the transition probability), is determined completely by the immediately previous l symbols $s_{i1}, \cdots, s_{il}$. That is,

IPR2021-00827
Unified EX1008 Page 157

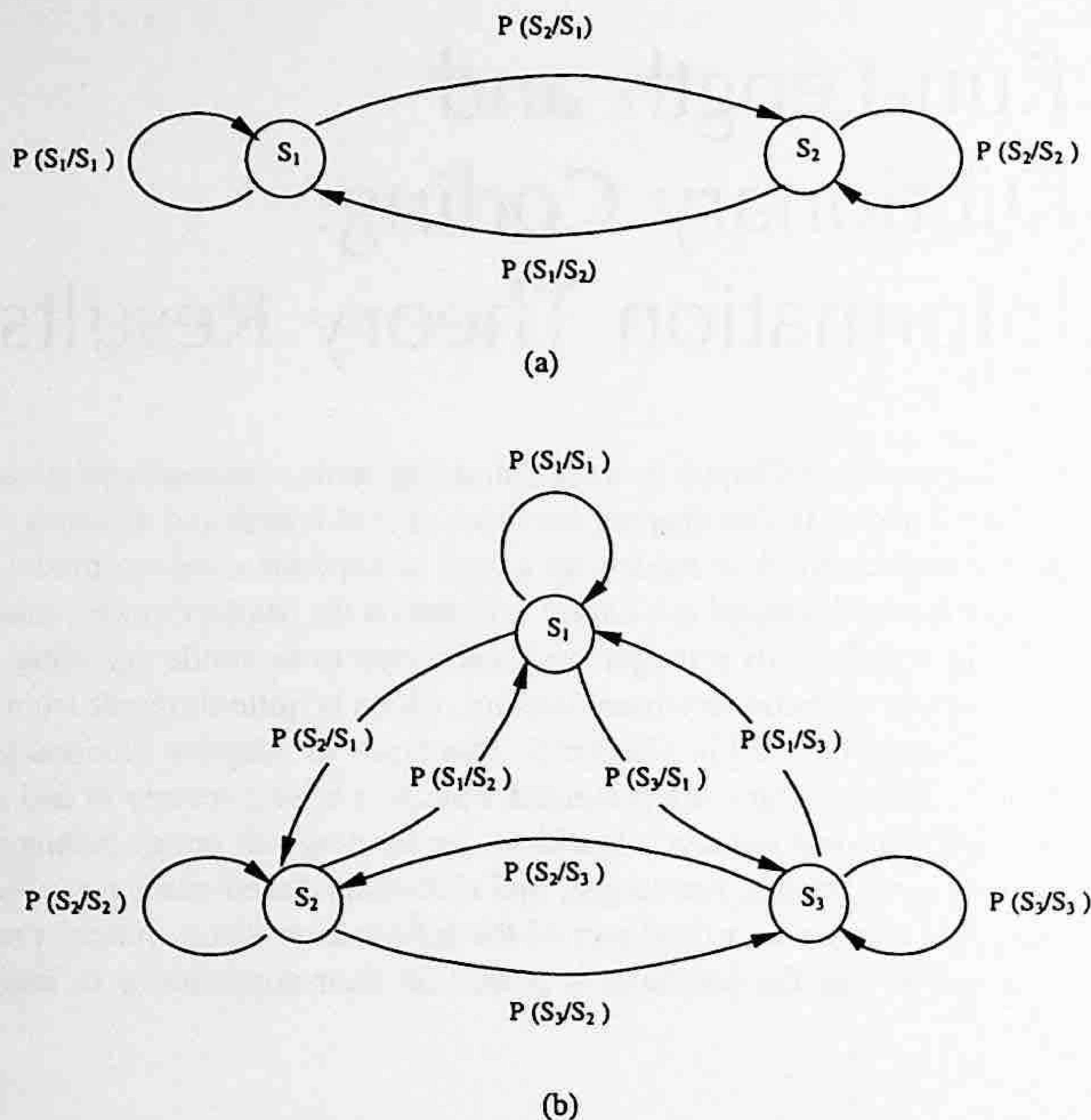


FIGURE 6.1 State diagrams of the first-order Markov sources with their source alphabets having (a) two symbols and (b) three symbols.

the knowledge of the entire sequence of previous symbols is equivalent to that of the l symbols immediately preceding the current symbol s_j.

An lth order Markov source can be described by what is called a *state diagram.* A state is a sequence of $(s_{i1}, s_{i2}, \cdots, s_{il})$ with $i1, i2, \cdots, il \in \{1,2,\cdots,m\}$. That is, any group of l symbols from the m symbols in the source alphabet S forms a state. When $l = 1$, it is called a first-order Markov source. The state diagrams of the first-order Markov sources, with their source alphabets having two and three symbols, are shown in Figure 6.1(a) and (b), respectively. Obviously, an lth order Markov source with m symbols in the source alphabet has a total of m^l different states. Therefore, we conclude that a state diagram consists of all the m^l states. In the diagram, all the transition probabilities together with appropriate arrows are used to indicate the state transitions.

The source entropy at a state $(s_{i1}, s_{i2}, \cdots, s_{il})$ is defined as

$$H\left(S\middle|s_{i1}, s_{i2}, \cdots, s_{il}\right) = -\sum_{j=1}^{m} p\left(s_j\middle|s_{i1}, s_{i2}, \cdots, s_{il}\right)\log_2 p\left(s_j\middle|s_{i1}, s_{i2}, \cdots, s_{il}\right) \tag{6.2}$$

The source entropy is defined as the statistical average of the entropy at all the states. That is,

$$H(S) = \sum_{(s_{i1}, s_{i2}, \cdots, s_{il}) \in S^l} p\left(s_{i1}, s_{i2}, \cdots, s_{il}\right) H\left(S\middle|s_{i1}, s_{i2}, \cdots, s_{il}\right), \tag{6.3}$$

IPR2021-00827
Unified EX1008 Page 158

where, as defined in the previous chapter, S^l denotes the lth extension of the source alphabet S. That is, the summation is carried out with respect to all l-tuples taking over the S^l. Extensions of a Markov source are defined below.

6.1.2 Extensions of a Discrete Markov Source

An extension of a Markov source can be defined in a similar way to that of an extension of a memoryless source in the previous chapter. The definition of extensions of a Markov source and the relation between the entropy of the original Markov source and the entropy of the nth extension of the Markov source are presented below without derivation. For the derivation, readers are referred to (Abramson, 1963).

6.1.2.1 Definition

Consider an lth order Markov source $S = \{s_1, s_2, \cdots, s_m\}$ and a set of conditional probabilities $p(s_j | s_{i1}, s_{i2}, \cdots, s_{il})$, where $j, i1, i2, \cdots, il \in \{1,2,\cdots,m\}$. Similar to the memoryless source discussed in Chapter 5, if n symbols are grouped into a block, then there is a total of m^n blocks. Each block can be viewed as a new source symbol. Hence, these m^n blocks form a new information source alphabet, called the nth extension of the source S and denoted by S^n. The nth extension of the lth-order Markov source is a kth-order Markov source, where k is the smallest integer greater than or equal to the ratio between l and n. That is,

$$k = \left\lceil \frac{l}{n} \right\rceil, \tag{6.4}$$

where the notation $\lceil a \rceil$ represents the operation of taking the smallest integer greater than or equal to the quantity a.

6.1.2.2 Entropy

Denote, respectively, the entropy of the lth order Markov source S by $H(S)$, and the entropy of the nth extension of the lth order Markov source, S^n, by $H(S^n)$. The following relation between the two entropies can be shown:

$$H(S^n) = nH(S) \tag{6.5}$$

6.1.3 Autoregressive (AR) Model

The Markov source discussed above represents a kind of dependence between source symbols in terms of the transition probability. Concretely, in determining the transition probability of a present source symbol given all the previous symbols, only the set of finitely many immediately preceding symbols matters. The autoregressive model is another kind of dependent source model that has been used often in image coding. It is defined below.

$$s_j = \sum_{k=1}^{l} a_k s_{ik} + x_j, \tag{6.6}$$

where s_j represents the currently observed source symbol, while s_{ik} with $k = 1,2,\cdots,l$ denote the l preceding observed symbols, a_k's are coefficients, and x_j is the current input to the model. If $l = 1$,

the model defined in Equation 6.6 is referred to as the first-order AR model. Clearly, in this case, the current source symbol is a linear function of its preceding symbol.

6.2 RUN-LENGTH CODING (RLC)

The term *run* is used to indicate the repetition of a symbol, while the term *run-length* is used to represent the number of repeated symbols, in other words, the number of consecutive symbols of the same value. Instead of encoding the consecutive symbols, it is obvious that encoding the run-length and the value that these consecutive symbols commonly share may be more efficient. According to an excellent early review on binary image compression by Arps (1979), RLC has been in use since the earliest days of information theory (Shannon and Weaver, 1949; Laemmel, 1951).

From the discussion of the JPEG in Chapter 4 (with more details in Chapter 7), it is seen that most of the DCT coefficients within a block of 8 × 8 are zero after certain manipulations. The DCT coefficients are zigzag scanned. The nonzero DCT coefficients and their addresses in the 8 × 8 block need to be encoded and transmitted to the receiver side. There, the nonzero DCT values are referred to as labels. The position information about the nonzero DCT coefficients is represented by the run-length of zeros between the nonzero DCT coefficients in the zigzag scan. The labels and the run-length of zeros are then Huffman coded.

Many documents such as letters, forms, and drawings can be transmitted using facsimile machines over the general switched telephone network (GSTN). In digital facsimile techniques, these documents are quantized into binary levels: black and white. The resolution of these binary tone images is usually very high. In each scan line, there are many consecutive white and black pixels, i.e., many alternate white runs and black runs. Therefore it is not surprising to see that RLC has proven to be efficient in binary document transmission. RLC has been adopted in the international standards for facsimile coding: the CCITT Recommendations T.4 and T.6.

RLC using only the horizontal correlation between pixels on the same scan line is referred to as 1-D RLC. It is noted that the first-order Markov source model with two symbols in the source alphabet depicted in Figure 6.1(a) can be used to characterize 1-D RLC. To achieve higher coding efficiency, 2-D RLC utilizes both horizontal and vertical correlation between pixels. Both the 1-D and 2-D RLC algorithms are introduced below.

6.2.1 1-D RUN-LENGTH CODING

In this technique, each scan line is encoded independently. Each scan line can be considered as a sequence of alternating, independent white runs and black runs. As an agreement between encoder and decoder, the first run in each scan line is assumed to be a white run. If the first actual pixel is black, then the run-length of the first white run is set to be zero. At the end of each scan line, there is a special codeword called end-of-line (EOL). The decoder knows the end of a scan line when it encounters an EOL codeword.

Denote run-length by r, which is integer-valued. All of the possible run-lengths construct a source alphabet R, which is a random variable. That is,

$$R = \{r : r \in 0, 1, 2, \cdots\} \tag{6.7}$$

Measurements on typical binary documents have shown that the maximum compression ratio, ζ_{max}, which is defined below, is about 25% higher when the white and black runs are encoded separately (Hunter and Robinson, 1980). The average white run-length, $\bar{r}_W$, can be expressed as

$$\bar{r}_W = \sum_{r=0}^{m} r \cdot P_W(r) \tag{6.8}$$

where m is the maximum value of the run-length, and $P_W(r)$ denotes the occurrence probability of a white run with length r. The entropy of the white runs, H_W, is

$$H_W = -\sum_{r=0}^{m} P_W(r)\log_2 P_W(r) \tag{6.9}$$

For the black runs, the average run-length $\bar{r}_B$ and the entropy H_B can be defined similarly. The maximum theoretical compression factor ζ_{max} is

$$\zeta_{max} = \frac{\bar{r}_W + \bar{r}_B}{H_W + H_B} \tag{6.10}$$

Huffman coding is then applied to two source alphabets. According to CCITT Recommendation T.4, A4 size (210 × 297 mm) documents should be accepted by facsimile machines. In each scan line, there are 1728 pixels. This means that the maximum run-length for both white and black runs is 1728, i.e., $m = 1728$. Two source alphabets of such a large size imply the requirement of two large codebooks, hence the requirement of large storage space. Therefore, some modification was made, resulting in the "modified" Huffman (MH) code.

In the modified Huffman code, if the run-length is larger than 63, then the run-length is represented as

$$r = M \times 64 + T \quad as \quad r > 63, \tag{6.11}$$

where M takes integer values from 1, 2 to 27, and $M \times 64$ is referred to as the makeup run-length; T takes integer values from 0, 1 to 63, and is called the terminating run-length. That is, if $r \le 63$, the run-length is represented by a terminating codeword only. Otherwise, if $r > 63$, the run-length is represented by a makeup codeword and a terminating codeword. A portion of the modified Huffman code table (Hunter and Robinson, 1980) is shown in Table 6.1. In this way, the requirement of large storage space is alleviated. The idea is similar to that behind modified Huffman coding, discussed in Chapter 5.

6.2.2 2-D Run-Length Coding

The 1-D run-length coding discussed above only utilizes correlation between pixels within a scan line. In order to utilize correlation between pixels in neighboring scan lines to achieve higher coding efficiency, 2-D run-length coding was developed. In Recommendation T.4, the modified relative element address designate (READ) code, also known as the modified READ code or simply the MR code, was adopted.

The modified READ code operates in a line-by-line manner. In Figure 6.2, two lines are shown. The top line is called the reference line, which has been coded, while the bottom line is referred to as the coding line, which is being coded. There are a group of five changing pixels, a_0, a_1, a_2, b_1, b_2, in the two lines. Their relative positions decide which of the three coding modes is used. The starting changing pixel a_0 (hence, five changing points) moves from left to right and from top to bottom as 2-D run-length coding proceeds. The five changing pixels and the three coding modes are defined below.

6.2.2.1 Five Changing Pixels

By a changing pixel, we mean the first pixel encountered in white or black runs when we scan an image line-by-line, from left to right, and from top to bottom. The five changing pixels are defined below.

IPR2021-00827
Unified EX1008 Page 161

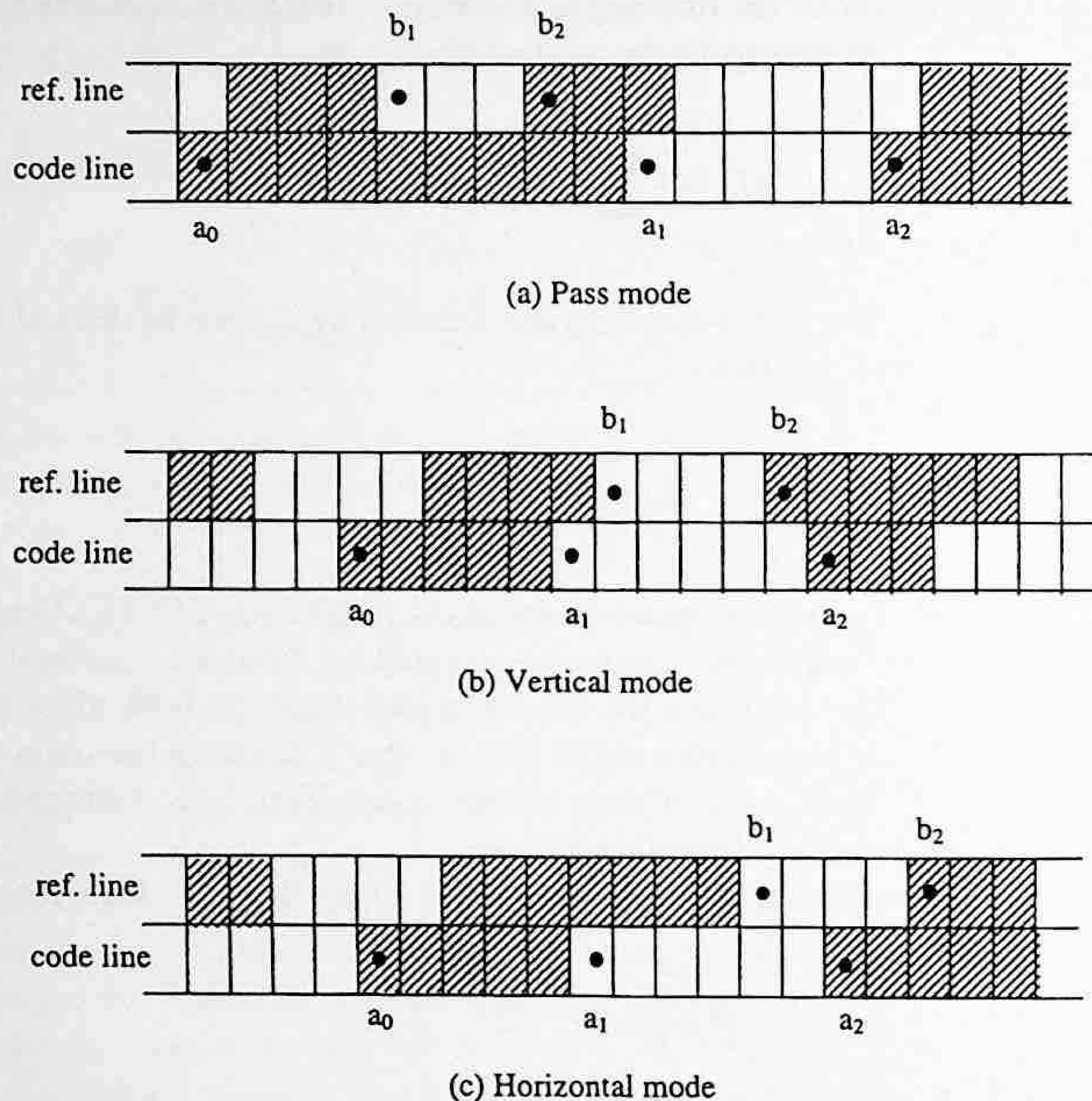


FIGURE 6.2 2-D run-length coding.

a_0: The reference-changing pixel in the coding line. Its position is defined in the previous coding mode, whose meaning will be explained shortly. At the beginning of a coding line, a_0 is an imaginary white changing pixel located before the first actual pixel in the coding line.

a_1: The next changing pixel in the coding line. Because of the above-mentioned left-to-right and top-to-bottom scanning order, it is at the right-hand side of a_0. Since it is a changing pixel, it has an opposite "color" to that of a_0.

a_2: The next changing pixel after a_1 in the coding line. It is to the right of a_1 and has the same color as that of a_0.

b_1: The changing pixel in the reference line that is closest to a_0 from the right and has the same color as a_1.

b_2: The next changing pixel in the reference line after b_1.

6.2.2.2 Three Coding Modes

Pass Coding Mode — If the changing pixel b_2 is located to the left of the changing pixel a_1, it means that the run in the reference line starting from b_1 is not adjacent to the run in the coding line starting from a_1. Note that these two runs have the same color. This is called the pass coding mode. A special codeword, "0001", is sent out from the transmitter. The receiver then knows that the run starting from a_0 in the coding line does not end at the pixel below b_2. This pixel (below b_2 in the coding line) is identified as the reference-changing pixel a_0 of the new set of five changing pixels for the next coding mode.

Vertical Coding Mode — If the relative distance along the horizontal direction between the changing pixels a_1 and b_1 is not larger than three pixels, the coding is conducted in vertical coding

IPR2021-00827
Unified EX1008 Page 162

TABLE 6.1
Modified Huffman Code Table (Hunter and Robinson, 1980)

Run-Length	White Runs	Black Runs
	Terminating Codewords	
0	00110101	0000110111
1	000111	010
2	0111	11
3	1000	10
4	1011	011
5	1100	0011
6	1110	0010
7	1111	00011
8	10011	000101
⋮	⋮	⋮
60	01001011	000000101100
61	00110010	000001011010
62	00110011	000001100110
63	00110100	000001100111
	Makeup Codewords	
64	11011	0000001111
128	10010	000011001000
192	010111	000011001001
256	0110111	000001011011
⋮	⋮	⋮
1536	010011001	0000001011010
1600	010011010	0000001011011
1664	011000	0000001100100
1728	010011011	0000001100101
EOL	000000000001	000000000001

TABLE 6.2
2-D Run-Length Coding Table

Mode	Conditions	Output Codeword	Position of New a_0
Pass coding mode	$b_2a_1 < 0$	0001	Under b_2 in coding line
Vertical coding mode	$a_1b_1 = 0$	1	a_1
	$a_1b_1 = 1$	011	
	$a_1b_1 = 2$	000011	
	$a_1b_1 = 3$	0000011	
	$a_1b_1 = -1$	010	
	$a_1b_1 = -2$	000010	
	$a_1b_1 = -3$	0000010	
Horizontal coding mode	$\lvert a_1b_1 \rvert > 3$	$001 + (a_0a_1) + (a_1a_2)$	a_2

Note: $\lvert x_iy_j \rvert$: distance between x_i and y_j, $x_iy_j > 0$: x_i is right to y_j, $x_iy_j < 0$: x_i is left to y_j. (x_iy_j): codeword of the run denoted by x_iy_j taken from the modified Huffman code.

Source: From Hunter and Robinson (1980).

mode. That is, the position of a_1 is coded with reference to the position of b_1. Seven different codewords are assigned to seven different cases: the distance between a_1 and b_1 equals 0, ±1, ±2, ±3, where + means a_1 is to the right of b_1, while – means a_1 is to the left of b_1. The a_1 then becomes the reference changing pixel a_0 of the new set of five changing pixels for the next coding mode.

Horizontal Coding Mode — If the relative distance between the changing pixels a_1 and b_1 is larger than three pixels, the coding is conducted in horizontal coding mode. Here, 1-D run-length coding is applied. Specifically, the transmitter sends out a codeword consisting the following three parts: a flag "001"; a 1-D run-length codeword for the run from a_0 to a_1; a 1-D run-length codeword for the run from a_1 to a_2. The a_2 then becomes the reference changing pixel a_0 of the new set of five changing pixels for the next coding mode. Table 6.2 contains three coding modes and the corresponding output codewords. There, (a_0a_1) and (a_1a_2) represent 1-D run-length codewords of run-length a_0a_1 and a_1a_2, respectively.

6.2.3 Effect of Transmission Error and Uncompressed Mode

In this subsection, effect of transmission error in the 1-D and 2-D RLC cases and uncompressed mode are discussed.

6.2.3.1 Error Effect in the 1-D RLC Case

As introduced above, the special codeword EOL is used to indicate the end of each scan line. With the EOL, 1-D run-length coding encodes each scan line independently. If a transmission error occurs in a scan line, there are two possibilities that the effect caused by the error is limited within the scan line. One possibility is that *resynchronization* is established after a few runs. One example is shown in Figure 6.3. There, the transmission error takes place in the second run from the left. Resynchronization is established in the fifth run in this example. Another possibility lies in the EOL, which forces resynchronization.

In summary, it is seen that the 1-D run-length coding will not propagate transmission error between scan lines. In other words, a transmission error will be restricted within a scan line. Although error detection and retransmission of data via an automatic repeat request (ARQ) system is supposed to be able to effectively handle the error susceptibility issue, the ARQ technique was not included in Recommendation T.4 due to the computational complexity and extra transmission time required.

Once the number of decoded pixels between two consecutive EOL codewords is not equal to 1728 (for an A4 size document), an error has been identified. Some *error concealment* techniques can be used to reconstruct the scan line (Hunter and Robinson, 1980). For instance, we can repeat

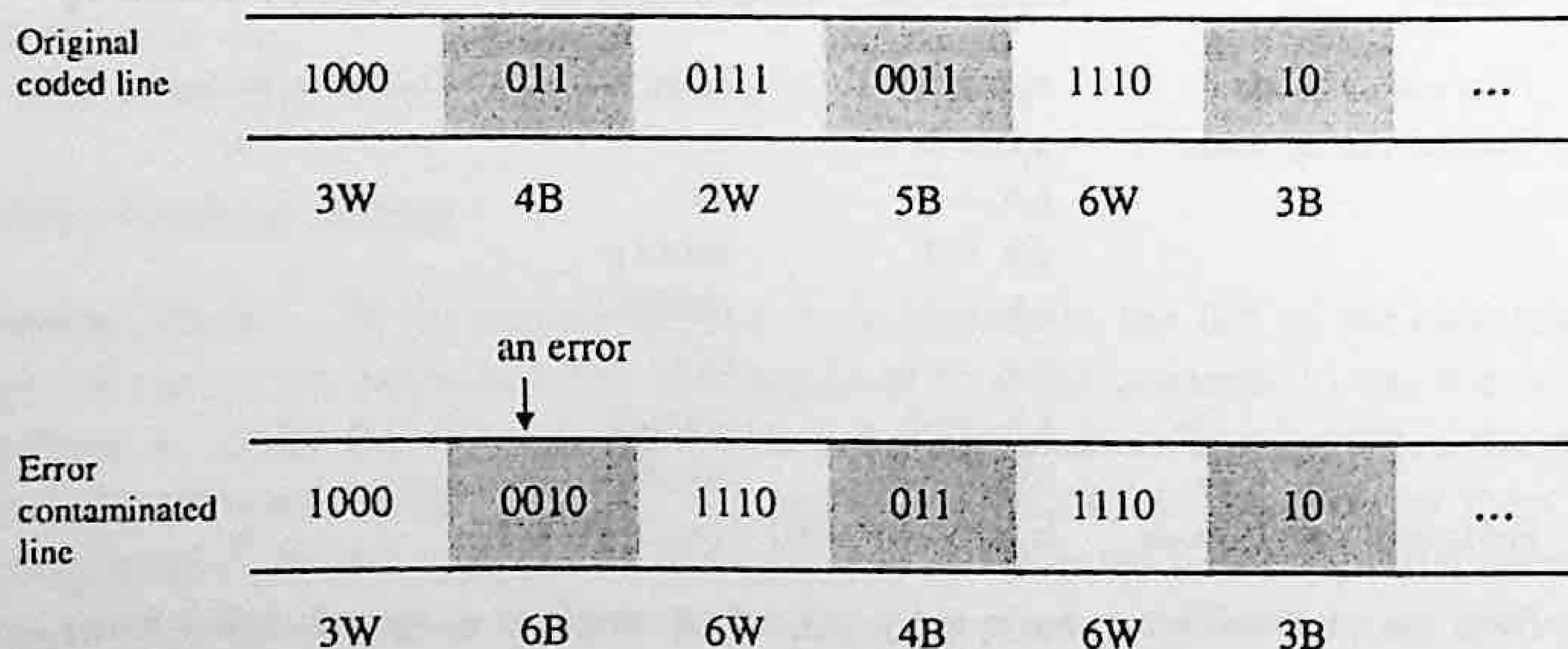


FIGURE 6.3 Establishment of resynchronization after a few runs.

IPR2021-00827
Unified EX1008 Page 164

the previous line, or replace the damaged line by a white line, or use a correlation technique to recover the line as much as possible.

6.2.3.2 Error Effect in the 2-D RLC Case

From the above discussion, we realize that 2-D RLC is more efficient than 1-D RLC on the one hand. The 2-D RLC is more susceptible to transmission errors than the 1-D RLC on the other hand. To prevent error propagation, there is a parameter used in 2-D RLC, known as the *K-factor*, which specifies the number of scan lines that are 2-D RLC coded.

Recommendation T.4 defined that no more than K-1 consecutive scan lines be 2-D RLC coded after a 1-D RLC coded line. For binary documents scanned at normal resolution, K = 2. For documents scanned at high resolution, K = 4.

According to Arps (1979), there are two different types of algorithms in binary image coding, *raster* algorithms and *area* algorithms. Raster algorithms only operate on data within one or two raster scan lines. They are hence mainly 1-D in nature. Area algorithms are truly 2-D in nature. They require that all, or a substantial portion, of the image is in random access memory. From our discussion above, we see that both 1-D and 2-D RLC defined in T.4 belong to the category of raster algorithms. Area algorithms require large memory space and are susceptible to transmission noise.

6.2.3.3 Uncompressed Mode

For some detailed binary document images, both 1-D and 2-D RLC may result in data expansion instead of data compression. Under these circumstances the number of coding bits is larger than the number of bilevel pixels. An uncompressed mode is created as an alternative way to avoid data expansion. Special codewords are assigned for the uncompressed mode.

For the performances of 1-D and 2-D RLC applied to eight CCITT test document images, and issues such as "fill bits" and "minimum scan line time (MSLT)," to name only a few, readers are referred to an excellent tutorial paper by Hunter and Robinson (1980).

6.3 DIGITAL FACSIMILE CODING STANDARDS

Facsimile transmission, an important means of communication in modern society, is often used as an example to demonstrate the mutual interaction between widely used applications and standardization activities. Active facsimile applications and the market brought on the necessity for international standardization in order to facilitate interoperability between facsimile machines worldwide. Successful international standardization, in turn, has stimulated wider use of facsimile transmission and, hence, a more demanding market. Facsimile has also been considered as a major application for binary image compression.

So far, facsimile machines are classified in four different groups. Facsimile apparatuses in groups 1 and 2 use analog techniques. They can transmit an A4 size (210 × 297 mm) document scanned at 3.85 lines/mm in 6 and 3 min, respectively, over the GSTN. International standards for these two groups of facsimile apparatus are CCITT (now ITU) Recommendations T.2 and T.3, respectively. Group 3 facsimile machines use digital techniques and hence achieve high coding efficiency. They can transmit the A4 size binary document scanned at a resolution of 3.85 lines/mm and sampled at 1728 pixels per line in about 1 min at a rate of 4800 b/sec over the GSTN. The corresponding international standard is CCITT Recommendations T.4. Group 4 facsimile apparatuses have the same transmission speed requirement as that for group 3 machines, but the coding technique is different. Specifically, the coding technique used for group 4 machines is based on 2-D run-length coding, discussed above, but modified to achieve higher coding efficiency. Hence it is referred to as the modified modified READ coding, abbreviated MMR. The corresponding standard is CCITT Recommendations T.6. Table 6.3 summarizes the above descriptions.

TABLE 6.3 FACSIMILE CODING STANDARDS

Group of Facsimile Apparatuses	Speed Requirement for A-4 Size Document	Analog or Digital Scheme	CCITT Recommendation	Compression Technique		
				Model	Basic Coder	Algorithm Acronym
G_1	6 min	Analog	T.2	—	—	—
G_2	3 min	Analog	T.3	—	—	—
G_3	1 min	Digital	T.4	1-D RLC	Modified Huffman	MH
				2-D RLC (optional)		MR
G_4	1 min	Digital	T.6	2-D RLC	Modified Huffman	MMR

6.4 DICTIONARY CODING

Dictionary coding, the focus of this section, is different from Huffman coding and arithmetic coding, discussed in the previous chapter. Both Huffman and arithmetic coding techniques are based on a statistical model, and the occurrence probabilities play a particular important role. Recall that in the Huffman coding the shorter codewords are assigned to more frequently occurring source symbols. In dictionary-based data compression techniques a symbol or a string of symbols generated from a source alphabet is represented by an index to a dictionary constructed from the source alphabet. A dictionary is a list of symbols and strings of symbols. There are many examples of this in our daily lives. For instance, the string "September" is sometimes represented by an index "9," while a social security number represents a person in the U.S.

Dictionary coding is widely used in text coding. Consider English text coding. The source alphabet includes 26 English letters in both lower and upper cases, numbers, various punctuation marks, and the space bar. Huffman or arithmetic coding treats each symbol based on its occurrence probability. That is, the source is modeled as a memoryless source. It is well known, however, that this is not true in many applications. In text coding, *structure* or *context* plays a significant role. As mentioned earlier, it is very likely that the letter *u* appears after the letter *q*. Likewise, it is likely that the word "concerned" will appear after "As far as the weather is." The strategy of the dictionary coding is to build a dictionary that contains frequently occurring symbols and string of symbols. When a symbol or a string is encountered and it is contained in the dictionary, it is encoded with an index to the dictionary. Otherwise, if not in the dictionary, the symbol or the string of symbols is encoded in a less efficient manner.

6.4.1 Formulation of Dictionary Coding

To facilitate further discussion, we define dictionary coding in a precise manner (Bell et al., 1990). We denote a source alphabet by S. A dictionary consisting of two elements is defined as $D = (P, C)$, where P is a finite set of phrases generated from the S, and C is a coding function mapping P onto a set of codewords.

The set P is said to be complete if any input string can be represented by a series of phrases chosen from the P. The coding function C is said to obey the prefix property if there is no codeword that is a prefix of any other codeword. For practical usage, i.e., for reversible compression of any input text, the phrase set P must be complete and the coding function C must satisfy the prefix property.

6.4.2 Categorization of Dictionary-Based Coding Techniques

The heart of dictionary coding is the formulation of the dictionary. A successfully built dictionary results in data compression; the opposite case may lead to data expansion. According to the ways

in which dictionaries are constructed, dictionary coding techniques can be classified as static or adaptive.

6.4.2.1 Static Dictionary Coding

In some particular applications, the knowledge about the source alphabet and the related strings of symbols, also known as phrases, is sufficient for a fixed dictionary to be produced before the coding process. The dictionary is used at both the transmitting and receiving ends. This is referred to as static dictionary coding. The merit of the static approach is its simplicity. Its drawbacks lie in its relatively lower coding efficiency and less flexibility compared with adaptive dictionary techniques. By less flexibility, we mean that a dictionary built for a specific application is not normally suitable for utilization in other applications.

An example of static algorithms occurring is *digram* coding. In this simple and fast coding technique, the dictionary contains all source symbols and some frequently used pairs of symbols. In encoding, two symbols are checked at once to see if they are in the dictionary. If so, they are replaced by the index of the two symbols in the dictionary, and the next pair of symbols is encoded in the next step. If not, then the index of the first symbol is used to encode the first symbol. The second symbol is combined with the third symbol to form a new pair, which is encoded in the next step.

The digram can be straightforwardly extended to *n-gram*. In the extension, the size of the dictionary increases and so does its coding efficiency.

6.4.2.2 Adaptive Dictionary Coding

As opposed to the static approach, with the adaptive approach a completely defined dictionary does not exist prior to the encoding process and the dictionary is not fixed. At the beginning of coding, only an initial dictionary exists. It adapts itself to the input during the coding process. All the adaptive dictionary coding algorithms can be traced back to two different original works by Ziv and Lempel (1977, 1978). The algorithms based on Ziv and Lempel (1977) are referred to as the LZ77 algorithms, while those based on their 1978 work are the LZ78 algorithms. Prior to introducing the two landmark works, we will discuss the parsing strategy.

6.4.3 Parsing Strategy

Once we have a dictionary, we need to examine the input text and find a string of symbols that matches an item in the dictionary. Then the index of the item to the dictionary is encoded. This process of segmenting the input text into disjoint strings (whose union equals the input text) for coding is referred to as *parsing*. Obviously, the way to segment the input text into strings is not unique.

In terms of the highest coding efficiency, optimal parsing is essentially a shortest-path problem (Bell et al., 1990). In practice, however, a method called *greedy* parsing is used most often. In fact, it is used in all the LZ77 and LZ78 algorithms. With greedy parsing, the encoder searches for the longest string of symbols in the input that matches an item in the dictionary at each coding step. Greedy parsing may not be optimal, but it is simple in its implementation.

Example 6.1

Consider a dictionary, D, whose phrase set $P = \{a, b, ab, ba, bb, aab, bbb\}$. The codewords assigned to these strings are $C(a) = 10$, $C(b) = 11$, $C(ab) = 010$, $C(ba) = 0101$, $C(bb) = 01$, $C(abb) = 11$, and $C(bbb) = 0110$. Now the input text is *abbaab*.

Using greedy parsing, we then encode the text as $C(ab).C(ba).C(ab)$, which is a 10-bit string: 010.0101.010. In the above representations, the periods are used to indicate the division of segments in the parsing. This, however, is not an optimum solution. Obviously, the following parsing will be more efficient, i.e., $C(a).C(bb).C(aab)$, which is a 6-bit string: 10.01.11.

6.4.4 Sliding Window (LZ77) Algorithms

As mentioned earlier, LZ77 algorithms are a group of adaptive dictionary coding algorithms rooted in the pioneering work of Ziv and Lempel (1977). Since they are adaptive, there is no complete and fixed dictionary before coding. Instead, the dictionary changes as the input text changes.

6.4.4.1 Introduction

In the LZ77 algorithms, the dictionary used is actually a portion of the input text, which has been recently encoded. The text that needs to be encoded is compared with the strings of symbols in the dictionary. The longest matched string in the dictionary is characterized by a *pointer* (sometimes called a *token*), which is represented by a triple of data items. Note that this triple functions as an index to the dictionary, as mentioned above. In this way, a variable-length string of symbols is mapped to a fixed-length pointer.

There is a sliding window in the LZ77 algorithms. The window consists of two parts: a search buffer and a look-ahead buffer. The search buffer contains the portion of the text stream that has recently been encoded which, as mentioned, is the dictionary; while the look-ahead buffer contains the text to be encoded next. The window slides through the input text stream from beginning to end during the entire encoding process. This explains the term *sliding* window. The size of the search buffer is much larger than that of the look-ahead buffer. This is expected because what is contained in the search buffer is in fact the adaptive dictionary. The sliding window is usually on the order of a few thousand symbols, whereas the look-ahead buffer is on the order of several tens to one hundred symbols.

6.4.4.2 Encoding and Decoding

Below we present more details about the sliding window dictionary coding technique, i.e., the LZ77 approach, via a simple illustrative example.

Example 6.2
Figure 6.4 shows a sliding window. The input text stream is *ikaccbadaccbaccbaccgikmoabc*. In part (a) of the figure, a search buffer of nine symbols and a look-ahead buffer of six symbols are shown. All the symbols in the search buffer, *accbadacc*, have just been encoded. All the symbols in the look-ahead buffer, *baccba*, are to be encoded. (It is understood that the symbols before the

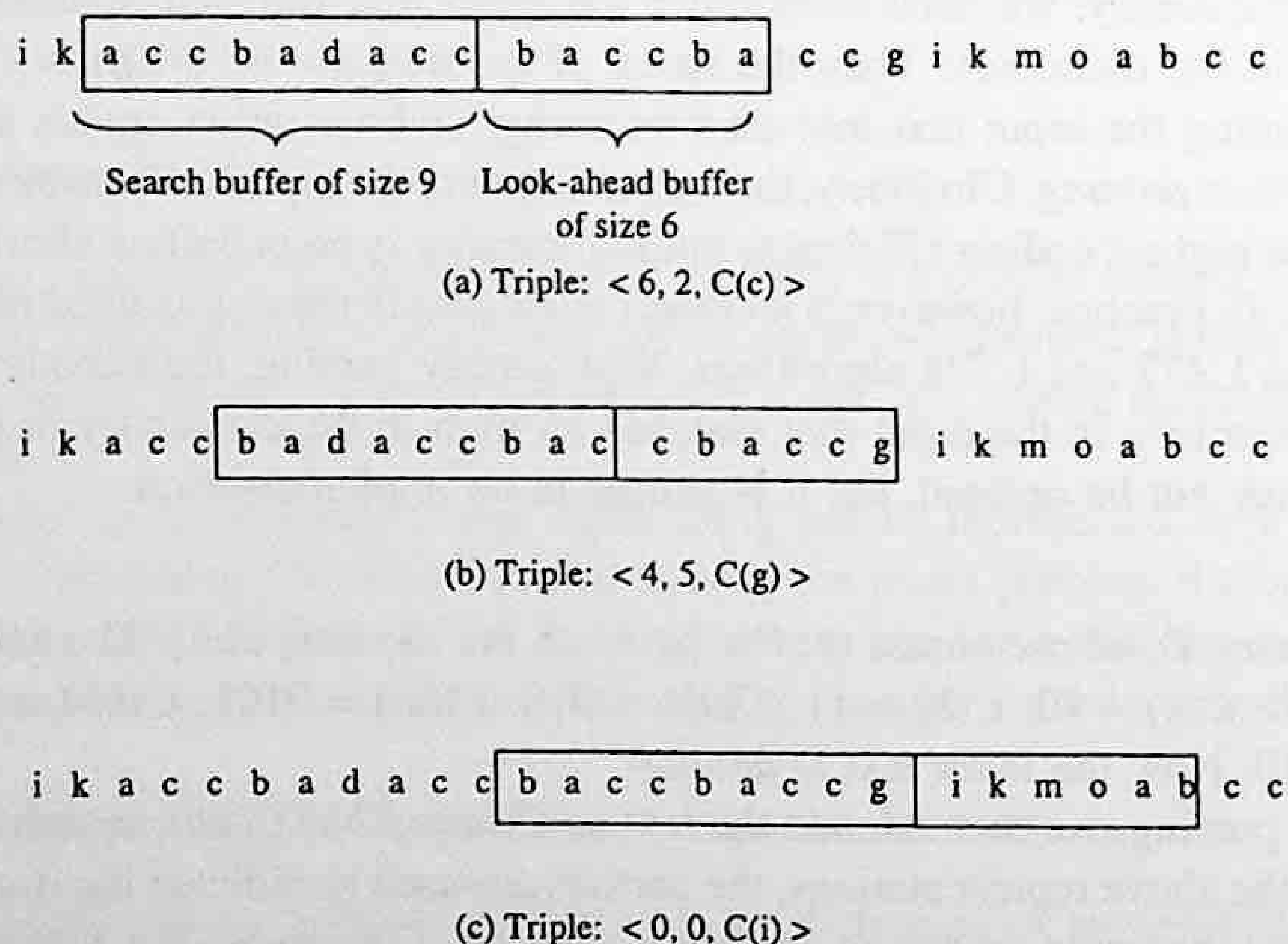


FIGURE 6.4 An encoding example using LZ77.

IPR2021-00827
Unified EX1008 Page 168

search buffer have been encoded and the symbols after the look-ahead buffer are to be encoded.) The strings of symbols, *ik* and *ccgikmoabcc*, are not covered by the sliding window at the moment.

At the moment, or in other words, in the first step of encoding, the symbol(s) to be encoded begin(s) with the symbol *b*. The pointer starts searching for the symbol *b* from the last symbol in the search buffer, *c*, which is immediately to the left of the first symbol *b* in the look-ahead buffer. It finds a match at the sixth position from *b*. It further determines that the longest string of the match is *ba*. That is, the maximum matching length is two. The pointer is then represented by a triple, <i,j,k>. The first item, "i", represents the distance between the first symbol in the look-ahead buffer and the position of the pointer (the position of the first symbol of the matched string). This distance is called *offset*. In this step, the offset is six. The second item in the triple, "j", indicates the length of the matched string. Here, the length of the matched string *ba* is two. The third item, "k", is the codeword assigned to the symbol immediately following the matched string in the look-ahead buffer. In this step, the third item is C(c), where C is used to represent a function to map symbol(s) to a codeword, as defined in Section 6.4.1. That is, the resulting triple after the first step is: <6, 2, C(c)>.

The reason to include the third item "k" into the triple is as follows. In the case where there is no match in the search buffer, both "i" and "j" will be zero. The third item at this moment is the codeword of the first symbol in the look-ahead buffer itself. This means that even in the case where we cannot find a match string, the sliding window still works. In the third step of the encoding process described below, we will see that the resulting triple is: <0, 0, C(i)>. The decoder hence understands that there is no matching, and the single symbol *i* is decoded.

The second step of the encoding is illustrated in part (b) of Figure 6.4. The sliding window has been shifted to the right by three positions. The first symbol to be encoded now is *c*, which is the left-most symbol in the look-ahead buffer. The search pointer moves towards the left from the symbol *c*. It first finds a match in the first position with a length of one. It then finds another match in the fourth position from the first symbol in the look-ahead buffer. Interestingly, the maximum matching can exceed the boundary between the search buffer and the look-ahead buffer and can enter the look-ahead buffer. Why this is possible will be seen shortly, when we discuss the decoding process. In this manner, it is found that the maximum length of matching is five. The last match is found at the fifth position. The length of the matched string, however, is only one. Since greedy parsing is used, the match with a length five is chosen. That is, the offset is four and the maximum match length is five. Consequently, the triple resulting from the second step is <4, 5, C(g)>.

The sliding window is then shifted to the right by six positions. The third step of the encoding is depicted in Part (c). Obviously, there is no matching of *i* in the search buffer. The resulting triple is hence <0, 0, C(i)>.

The encoding process can continue in this way. The possible cases we may encounter in the encoding, however, are described in the first three steps. Hence we end our discussion of the encoding process and discuss the decoding process. Compared with the encoding, the decoding is simpler because there is no need for matching, which involves many comparisons between the symbols in the look-ahead buffer and the symbols in the search buffer. The decoding process is illustrated in Figure 6.5.

In the above three steps, the resulting triples are <6, 2, C(c)>, <4, 5, C(g)>, and <0, 0, C(i)>. Now let us see how the decoder works. That is, how the decoder recovers the string *baccbaccgi* from these three triples.

In part (a) of Figure 6.5, the search buffer is the same as that in part (a) of Figure 6.4. That is, the string *accbadacc* stored in the search window is what was just decoded.

Once the first triple <6, 2, C(c)> is received, the decoder will move the decoding pointer from the first position in the look-ahead buffer to the left by six positions. That is, the pointer will point to the symbol *b*. The decoder then copies the two symbols starting from *b*, i.e., *ba*, into the look-ahead buffer. The symbol *c* will be copied right to *ba*. This is shown in part (b) of Figure 6.5. The window is then shifted to the right by three positions, as shown in part (c) of Figure 6.5.

IPR2021-00827
Unified EX1008 Page 169

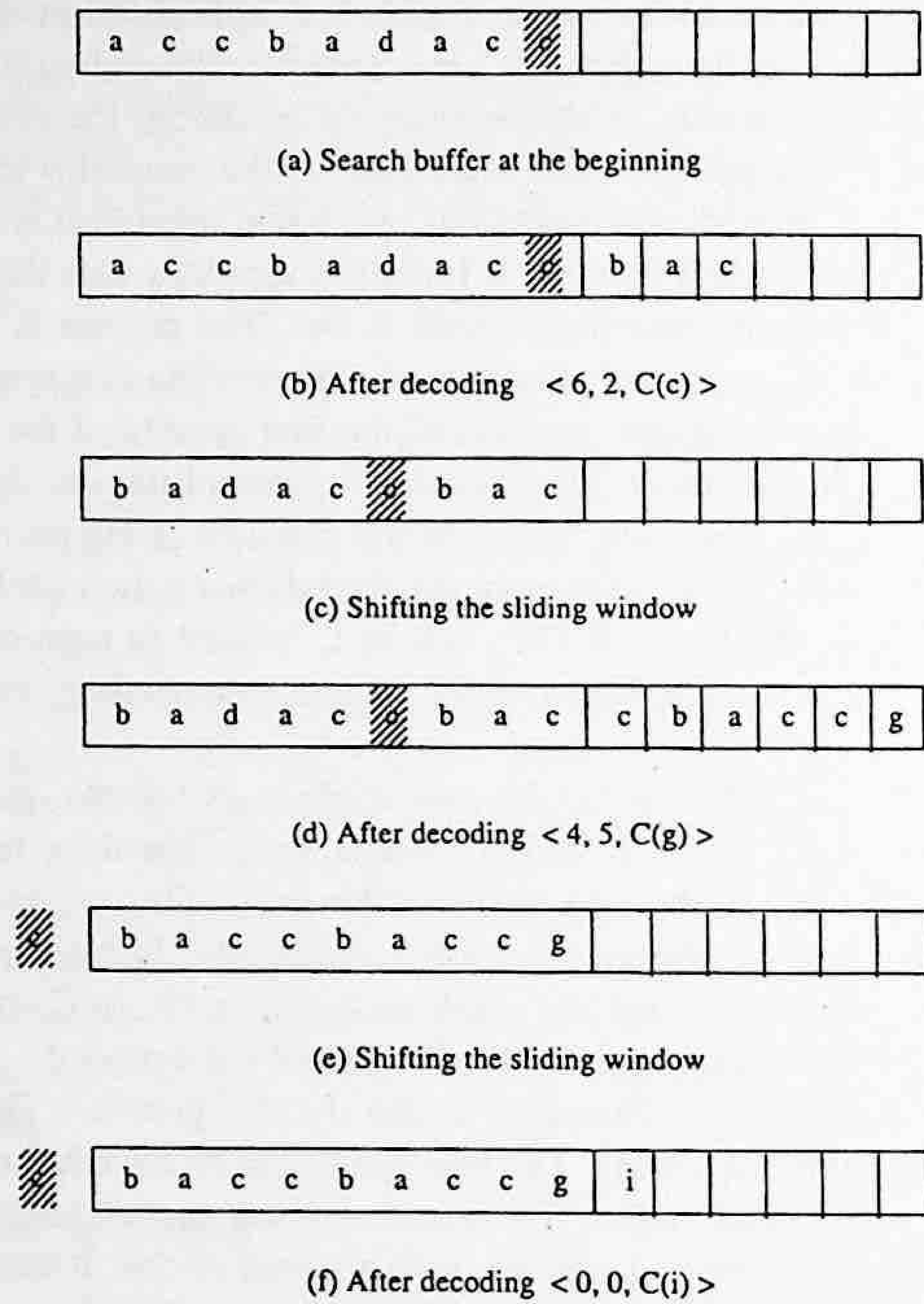


FIGURE 6.5 A decoding example using LZ77.

After the second triple <4, 5, C(g)> is received, the decoder moves the decoding pointer from the first position of the look-ahead buffer to the left by four positions. The pointer points to the symbol *c*. The decoder then copies five successive symbols starting from the symbol *c* pointed by the pointer. We see that at the beginning of this copying process there are only four symbols available for copying. Once the first symbol is copied, however, all five symbols are available. After copying, the symbol *g* is added to the end of the five copied symbols in the look-ahead buffer. The results are shown in part (c) of Figure 6.5. Part (d) then shows the window shifting to the right by six positions.

After receiving the triple <0, 0, C(i)>, the decoder knows that there is no match and a single symbol *i* is encoded. Hence, the decoder adds the symbol *i* following the symbol *g*. This is shown in part (f) of Figure 6.5.

In Figure 6.5, for each part, the last previously encoded symbol *c* prior to the receiving of the three triples is shaded. From part (f), we see that the string added after the symbol *c* due to the three triples is *baccbaccgi*. This agrees with the sequence mentioned at the beginning of our discussion about the decoding process. We thus conclude that the decoding process has correctly decoded the encoded sequence from the last encoded symbol and the received triples.

6.4.4.3 Summary of the LZ77 Approach

The sliding window consists of two parts: the search buffer and the look-ahead buffer. The most recently encoded portion of the input text stream is contained in the search buffer, while the portion of the text that needs to be encoded immediately is in the look-ahead buffer. The first symbol in the look-ahead buffer, located to the right of the boundary between the two buffers, is the symbol

IPR2021-00827
Unified EX1008 Page 170

or the beginning of a string of symbols to be encoded at the moment. Let us call it the symbol *s*. The size of the search buffer is usually much larger than that of the look-ahead buffer.

In encoding, the search pointer moves to the left, away from the symbol *s*, to find a match of the symbol *s* in the search buffer. Once a match is found, the encoding process will further determine the length of the matched string. When there are multiple matches, the match that produces the longest matched string is chosen. The match is denoted by a triple <i, j, k>. The first item in the triple, "i", is the offset, which is the distance between the pointer pointing to the symbol giving the maximum match and the symbol *s*. The second item, "j", is the length of the matched string. The third item, "k", is the codeword of the symbol following the matched string in the look-ahead buffer. The sliding window is then shifted to the right by j+1 positions before the next coding step takes place.

When there is no matching in the search buffer, the triple is represented by <0, 0, C(s)>, where C(s) is the codeword assigned to the symbol *s*. The sliding window is then shifted to the right by one position.

The sliding window is shifted along the input text stream during the encoding process. The symbol *s* moves from the beginning symbol to the ending symbol of the input text stream.

At the very beginning, the content of the search buffer can be arbitrarily selected. For instance, the symbols in the search buffer may all be the space symbol.

Let us denote the size of the search buffer by *SB*, the size of the look-ahead buffer by *L*, and the size of the source alphabet by *A*. Assume that the natural binary code is used. Then we see that the LZ77 approach encodes variable-length strings of symbols with fixed-length codewords. Specifically, the offset "i" is of coding length $\lceil \log_2 SB \rceil$, the length of matched string "j" is of coding length $\lceil \log_2 (SB + L) \rceil$, and the codeword "k" is of coding length $\lceil \log_2 (A) \rceil$, where the sign $\lceil a \rceil$ denotes the smallest integer larger than *a*.

The length of the matched string is equal to $\lceil \log_2 (SB + L) \rceil$ because the search for the maximum matching can enter into the look-ahead buffer as shown in Example 6.2.

The decoding process is simpler than the encoding process since there are no comparisons involved in the decoding.

The most recently encoded symbols in the search buffer serve as the dictionary used in the LZ77 approach. The merit of doing so is that the dictionary is well adapted to the input text. The limitation of the approach is that if the distance between the repeated patterns in the input text stream is larger than the size of the search buffer, then the approach cannot utilize the structure to compress the text. A vivid example can be found in (Sayood, 1996).

A window with a moderate size, say, $SB + L \leq 8192$, can compress a variety of texts well. Several reasons have been analyzed by Bell et al. (1990).

Many variations have been made to improve coding efficiency of the LZ77 approach. The LZ77 produces a triple in each encoding step; i.e., the offset (position of the matched string), the length of the matched string, and the codeword of the symbol following the matched string. The transmission of the third item in each coding step is not efficient. This is true especially at the beginning of coding. A variant of the LZ77, referred to as the LZSS algorithm, improves this inefficiency.

6.4.5 LZ78 Algorithms

6.4.5.1 Introduction

As mentioned above, the LZ77 algorithms use a sliding window of fixed size, and both the search buffer and the look-ahead buffer have a fixed size. This means that if the distance between two repeated patterns is larger than the size of the search buffer, the LZ77 algorithms cannot work efficiently. The fixed size of both the buffers implies that the matched string cannot be longer than the sum of the sizes of the two buffers, placing another limitation on coding efficiency. Increasing the sizes of the search buffer and the look-ahead buffer seemingly will resolve the problem. A close

IPR2021-00827
Unified EX1008 Page 171

look, however, reveals that it also leads to increases in the number of bits required to encode the offset and matched string length, as well as an increase in processing complexity.

The LZ78 algorithms (Ziv and Lempel, 1978) eliminate the use of the sliding window. Instead, these algorithms use the encoded text as a dictionary which, potentially, does not have a fixed size. Each time a pointer (token) is issued, the encoded string is included in the dictionary. Theoretically, the LZ78 algorithms reach optimal performance as the encoded text stream approaches infinity. In practice, however, as mentioned above with respect to the LZ77, a very large dictionary will affect coding efficiency negatively. Therefore, once a preset limit to the dictionary size has been reached, either the dictionary is fixed for the future (if the coding efficiency is good), or it is reset to zero, i.e., it must be restarted.

Instead of the triples used in the LZ77, only pairs are used in the LZ78. Specifically, only the position of the pointer to the matched string and the symbol following the matched string need to be encoded. The length of the matched string does not need to be encoded since both the encoder and the decoder have exactly the same dictionary, i.e., the decoder knows the length of the matched string.

6.4.5.2 Encoding and Decoding

Like the discussion of the LZ77 algorithms, we will go through an example to describe the LZ78 algorithms.

Example 6.3

Consider the text stream: *baccbaccacbcabccbbacc*. Table 6.4 shows the coding process. We see that for the first three symbols there is no match between the individual input symbols and the entries in the dictionary. Therefore, the doubles are, respectively, <0, C(b)>, <0, C(a)>, and <0, C(c)>, where 0 means no match, and C(b), C(a), and C(c) represent the codewords of *b*, *a*, and *c*, respectively. After symbols b, a, c, comes *c*, which finds a match in the dictionary (the third entry). Therefore, the next symbol *b* is combined to be considered. Since the string *cb* did not appear before, it is encoded as a double and it is appended as a new entry into the dictionary. The first item in the double is the index of the matched entry *c*, 3, the second item is the index/codeword of the symbol following the match *b*, 1. That is, the double is <3, 1>. The following input symbol is *a*, which appeared in the dictionary. Hence, the next symbol *c* is taken into consideration. Since the string *ac* is not an entry of the dictionary, it is encoded with a double. The first item in the double is the index of symbol *a*, 2; the second item is the index of symbol *c*, 3, i.e., <2, 3>. The encoding proceeds in this way. Take a look at Table 6.4. In general, as the encoding proceeds, the entries in the dictionary become longer and longer. First, entries with single symbols come out, but later, more and more entries with two symbols show up. After that, more and more entries with three symbols appear. This means that coding efficiency is increasing.

Now consider the decoding process. Since the decoder knows the rule applied in the encoding, it can reconstruct the dictionary and decode the input text stream from the received doubles. When the first double <0, C(b)> is received, the decoder knows that there is no match. Hence, the first entry in the dictionary is *b*. So is the first decoded symbol. From the second double <0, C(a)>, symbol *a* is known as the second entry in the dictionary as well as the second decoded symbol. Similarly, the next entry in the dictionary and the next decoded symbol are known as *c*. When the following double <3, 1> is received. The decoder knows from two items, 3 and 1, that the next two symbols are the third and the first entries in the dictionary. This indicates that the symbols *c* and *b* are decoded, and the string *cb* becomes the fourth entry in the dictionary.

We omit the next two doubles and take a look at the double <4, 3>, which is associated with Index 7 in Table 6.4. Since the first item in the double is 4, it means that the maximum matched string is *cb*, which is associated with Index 4 in Table 6.4. The second item in the double, 3, implies that the symbol following the match is the third entry *c*. Therefore the decoder decodes a string *cbc*. Also the string *cbc* becomes the seventh entry in the reconstructed dictionary. In this way, the

IPR2021-00827
Unified EX1008 Page 172

TABLE 6.4
An Encoding Example Using the LZ78 Algorithm

Index	Doubles	Encoded Symbols
1	< 0, C(b) >	b
2	< 0, C(a) >	a
3	< 0, C(c) >	c
4	< 3, 1 >	cb
5	< 2, 3 >	ac
6	< 3, 2 >	ca
7	< 4, 3 >	cbc
8	< 2, 1 >	ab
9	< 3, 3 >	cc
10	< 1, 1 >	bb
11	< 5, 3 >	acc

decoder can reconstruct the exact same dictionary as that established by the encoder and decode the input text stream from the received doubles.

6.4.5.3 LZW Algorithm

Both the LZ77 and LZ78 approaches, when published in 1977 and 1978, respectively, were theory oriented. The effective and practical improvement over the LZ78 by Welch (1984) brought much attention to the LZ dictionary coding techniques. The resulting algorithm is referred to the LZW algorithm. It removed the second item in the double (the index of the symbol following the longest matched string) and, hence, it enhanced coding efficiency. In other words, the LZW only sends the indexes of the dictionary to the decoder. For the purpose, the LZW first forms an initial dictionary, which consists of all the individual source symbols contained in the source alphabet. Then, the encoder examines the input symbol. Since the input symbol matches to an entry in the dictionary, its succeeding symbol is cascaded to form a string. The cascaded string does not find a match in the initial dictionary. Hence, the index of the matched symbol is encoded and the enlarged string (the matched symbol followed by the cascaded symbol) is listed as a new entry in the dictionary. The encoding process continues in this manner.

For the encoding and decoding processes, let us go through an example to see how the LZW algorithm can encode only the indexes and the decoder can still decode the input text string.

Example 6.4

Consider the following input text stream: *accbadaccbaccbacc*. We see that the source alphabet is $S = \{a,b,c,d,\}$. The top portion of Table 6.5 (with indexes 1,2,3,4) gives a possible initial dictionary used in the LZW. When the first symbol a is input, the encoder finds that it has a match in the dictionary. Therefore the next symbol c is taken to form a string ac. Because the string ac is not in the dictionary, it is listed as a new entry in the dictionary and is given an index, 5. The index of the matched symbol a, 1, is encoded. When the second symbol, c, is input the encoder takes the following symbol c into consideration because there is a match to the second input symbol c in the dictionary. Since the string cc does not match any existing entry, it becomes a new entry in the dictionary with an index, 6. The index of the matched symbol (the second input symbol), c, is encoded. Now consider the third input symbol c, which appeared in the dictionary. Hence, the following symbol b is cascaded to form a string cb. Since the string cb is not in the dictionary, it becomes a new entry in the dictionary and is given an index, 7. The index of matched symbol c, 3, is encoded. The process proceeds in this fashion.

IPR2021-00827
Unified EX1008 Page 173

TABLE 6.5
An Example of the Dictionary Coding Using the LZW Algorithm

Index	Entry	Input Symbols	Encoded Index
1	a	Initial dictionary	
2	b		
3	c		
4	d		
5	ac	a	1
6	cc	c	3
7	cb	c	3
8	ba	b	2
9	ad	a	1
10	da	d	4
11	acc	a,c	5
12	cba	c,b	7
13	accb	a,c,c	11
14	bac	b,a,	8
15	cc...	c,c,...	

Take a look at entry 11 in the dictionary shown in Table 6.5. The input symbol at this point is *a*. Since it has a match in the previous entries, its next symbol *c* is considered. Since the string *ac* appeared in entry 5, the succeeding symbol *c* is combined. Now the new enlarged string becomes *acc* and it does not have a match in the previous entries. It is thus added to the dictionary. And a new index, 11, is given to the string *acc*. The index of the matched string *ac*, 5, is encoded and transmitted. The final sequence of encoded indexes is 1, 3, 3, 2, 1, 4, 5, 7, 11, 8. Like the LZ78, the entries in the dictionary become longer and longer in the LZW algorithm. This implies high coding efficiency since long strings can be represented by indexes.

Now let us take a look at the decoding process to see how the decoder can decode the input text stream from the received index. Initially, the decoder has the same dictionary (the top four rows in Table 6.5) as that in the encoder. Once the first index 1 comes, the decoder decodes a symbol *a*. The second index is 3, which indicates that the next symbol is *c*. From the rule applied in encoding, the decoder knows further that a new entry *ac* has been added to the dictionary with an index 5. The next index is 3. It is known that the next symbol is also *c*. It is also known that the string *cc* has been added into the dictionary as the sixth entry. In this way, the decoder reconstructs the dictionary and decodes the input text stream.

6.4.5.4 Summary

The LZW algorithm, as a representative of the LZ78 approach, is summarized below.

The initial dictionary contains the indexes for all the individual source symbols. At the beginning of encoding, when a symbol is input, since it has a match in the initial dictionary, the next symbol is cascaded to form a two-symbol string. Since the two-symbol string cannot find a match in the initial dictionary, the index of the former symbol is encoded and transmitted, and the two-symbol string is added to the dictionary with a new, incremented index. The next encoding step starts with the latter symbol of the two.

In the middle, the encoding process starts with the last symbol of the latest added dictionary entry. Since it has a match in the previous entries, its succeeding symbol is cascaded after the symbol to form a string. If this string appeared before in the dictionary (i.e., the string finds a

IPR2021-00827
Unified EX1008 Page 174

match), the next symbol is cascaded as well. This process continues until such an enlarged string cannot find a match in the dictionary. At this moment, the index of the last matched string (the longest match) is encoded and transmitted, and the enlarged and unmatched string is added into the dictionary as a new entry with a new, incremented index.

Decoding is a process of transforming the index string back to the corresponding symbol string. In order to do so, however, the dictionary must be reconstructed in exactly the same way as that established in the encoding process. That is, the initial dictionary is constructed first in the same way as that in the encoding. When decoding the index string, the decoder reconstructs the same dictionary as that in the encoder according to the rule used in the encoding.

Specifically, at the beginning of the decoding, after receiving an index, a corresponding single symbol can be decoded. Via the next received index, another symbol can be decoded. From the rule used in the encoding, the decoder knows that the two symbols should be cascaded to form a new entry added into the dictionary with an incremented index. The next step in the decoding will start from the latter symbol among the two symbols.

Now consider the middle of the decoding process. The presently received index is used to decode a corresponding string of input symbols according to the reconstructed dictionary at the moment. (Note that this string is said to be with the present index.) It is known from the encoding rule that the symbols in the string associated with the next index should be considered. (Note that this string is said to be with the next index.) That is, the first symbol in the string with the next index should be appended to the last symbol in the string with the present index. The resultant combination, i.e., the string with the present index followed by the first symbol in the string with the next index, cannot find a match to an entry in the dictionary. Therefore, the combination should be added to the dictionary with an incremented index. At this moment, the next index becomes the new present index, and the index following the next index becomes the new next index. The decoding process then proceeds in the same fashion in a new decoding step.

Compared with the LZ78 algorithm, the LZW algorithm eliminates the necessity of having the second item in the double, an index/codeword of the symbol following a matched string. That is, the encoder only needs to encode and transmit the first item in the double. This greatly enhances the coding efficiency and reduces the complexity of the LZ algorithm.

6.4.5.5 Applications

The CCITT Recommendation V.42 bis is a data compression standard used in modems that connect computers with remote users via the GSTN. In the compressed mode, the LZW algorithm is recommended for data compression.

In image compression, the LZW finds its application as well. Specifically, it is utilized in the graphic interchange format (GIF) which was created to encode graphical images. GIF is now also used to encode natural images, though it is not very efficient in this regard. For more information, readers are referred to Sayood (1996). The LZW algorithm is also used in the UNIX Compress command.

6.5 INTERNATIONAL STANDARDS FOR LOSSLESS STILL IMAGE COMPRESSION

In the previous chapter, we studied Huffman and arithmetic coding techniques. We also briefly discussed the international standard for bilevel image compression, known as the JBIG. In this chapter, so far we have discussed another two coding techniques: the run-length and dictionary coding techniques. We also introduced the international standards for facsimile compression, in which the techniques known as the MH, MR, and MMR were recommended. All of these techniques involve lossless compression. In the next chapter, the international still image coding standard JPEG will be introduced. As we will see, the JPEG has four different modes. They can be divided into

IPR2021-00827
Unified EX1008 Page 175

two compression categories: lossy and lossless. Hence, we can talk about the lossless JPEG. Before leaving this chapter, however, we briefly discuss, compare, and summarize various techniques used in the international standards for lossless still image compression. For more details, readers are referred to an excellent survey paper by Arps and Truong (1994).

6.5.1 Lossless Bilevel Still Image Compression

6.5.1.1 Algorithms

As mentioned above, there are four different international standard algorithms falling into this category.

MH (Modified Huffman coding) — This algorithm is defined in CCITT Recommendation T.4 for facsimile coding. It uses the 1-D run-length coding technique followed by the "modified" Huffman coding technique.

MR (Modified READ [Relative Element Address Designate] coding) — Defined in CCITT Recommendation T.4 for facsimile coding. It uses the 2-D run-length coding technique followed by the "modified" Huffman coding technique.

MMR (Modified Modified READ coding) — Defined in CCITT Recommendation T.6. It is based on MR, but is modified to maximize compression.

JBIG (Joint Bilevel Image experts Group coding) — Defined in CCITT Recommendation T.82. It uses an adaptive 2-D coding model, followed by an adaptive arithmetic coding technique.

6.5.1.2 Performance Comparison

The JBIG test image set was used to compare the performance of the above-mentioned algorithms. The set contains scanned business documents with different densities, graphic images, digital halftones, and mixed (document and halftone) images.

Note that digital halftones, also named (digital) halftone images, are generated by using only binary devices. Some small black units are imposed on a white background. The units may assume different shapes: a circle, a square, and so on. The more dense the black units in a spot of an image, the darker the spot appears. The digital halftoning method has been used for printing gray-level images in newspapers and books. Digital halftoning through character overstriking, used to generate digital images in the early days for the experimental work associated with courses on digital image processing, has been described by Gonzalez and Woods (1992).

The following two observations on the performance comparison were made after the application of the several techniques to the JBIG test image set.

1. For bilevel images excluding digital halftones, the compression ratio achieved by these techniques ranges from 3 to 100. The compression ratio increases monotonically in the order of the following standard algorithms: MH, MR, MMR, JBIG.
2. For digital halftones, MH, MR, and MMR result in data expansion, while JBIG achieves compression ratios in the range of 5 to 20. This demonstrates that among the techniques, JBIG is the only one suitable for the compression of digital halftones.

6.5.2 Lossless Multilevel Still Image Compression

6.5.2.1 Algorithms

There are two international standards for multilevel still image compression:

JBIG (Joint Bilevel Image experts Group coding) — Defined in CCITT Recommendation T.82. It uses an adaptive arithmetic coding technique. To encode multilevel images, the JIBG decomposes multilevel images into bit-planes, then compresses these bit-planes using its bilevel

image compression technique. To further enhance the compression ratio, it uses Gary coding to represent pixel amplitudes instead of weighted binary coding.

JPEG (Joint Photographic (image) Experts Group coding) — Defined in CCITT Recommendation T.81. For lossless coding, it uses the differential coding technique. The predictive error is encoded using either Huffman coding or adaptive arithmetic coding techniques.

6.5.2.2 Performance Comparison

A set of color test images from the JPEG standards committee was used for performance comparison. The luminance component (Y) is of resolution 720 × 576 pixels, while the chrominance components (U and V) are of 360 × 576 pixels. The compression ratios calculated are the combined results for all the three components. The following observations have been reported.

1. When quantized in 8 bits per pixel, the compression ratios vary much less for multilevel images than for bilevel images, and are roughtly equal to 2.
2. When quantized with 5 bits per pixel down to 2 bits per pixel, compared with the lossless JPEG the JBIG achieves an increasingly higher compression ratio, up to a maximum of 29%.
3. When quantized with 6 bits per pixel, JBIG and lossless JPEG achieve similar compression ratios.
4. When quantized with 7 bits per pixel to 8 bits per pixel, the lossless JPEG achieves a 2.4 to 2.6% higher compression ratio than JBIG.

6.6 SUMMARY

Both Huffman coding and arithmetic coding, discussed in the previous chapter, are referred to as variable-length coding techniques, since the lengths of codewords assigned to different entries in a source alphabet are different. In general, a codeword of a shorter length is assigned to an entry with higher occurrence probabilities. They are also classified as fixed-length to variable-length coding techniques (Arps, 1979), since the entries in a source alphabet have the same fixed length. Run-length coding (RLC) and dictionary coding, the focus of this chapter, are opposite, and are referred to as variable-length to fixed-length coding techniques. This is because the runs in the RLC and the string in the dictionary coding are variable and are encoded with codewords of the same fixed length.

Based on RLC, the international standard algorithms for facsimile coding, MH, MR, and MMR have worked successfully except for dealing with digital halftones. That is, these algorithms result in data expansion when applied to digital halftones. The JBIG, based on an adaptive arithmetic coding technique, not only achieves a higher coding efficiency than MH, MR, and MMR for facsimile coding, but also compresses the digital halftones effectively.

Note that 1-D RLC utilizes the correlation between pixels within a scan line, whereas 2-D RLC utilizes the correlation between pixels within a few scan lines. As a result, 2-D RLC can obtain higher coding efficiency than 1-D RLC. On the other hand, 2-D RLC is more susceptible to transmission errors than 1-D RLC.

In text compression, the dictionary-based techniques have proven to be efficient. All the adaptive dictionary-based algorithms can be classified into two groups. One is based on a work by Ziv and Lempel in 1977, and another is based on their pioneering work in 1978. They are called the LZ77 and LZ78 algorithms, respectively. With the LZ77 algorithms, a fixed-size window slides through the input text stream. The sliding window consists of two parts: the search buffer and the look-ahead buffer. The search buffer contains the most recently encoded portion of the input text, while the look-ahead buffer contains the portion of the input text to be encoded immediately. For the symbols to be encoded, the LZ77 algorithms search for the longest match in the search buffer. The

IPR2021-00827
Unified EX1008 Page 177

information about the match: the distance between the matched string in the search buffer and that in the look-ahead buffer, the length of the matched string, and the codeword of the symbol following the matched string in the look-ahead buffer are encoded. Many improvements have been made in the LZ77 algorithms.

The performance of the LZ77 algorithms is limited by the sizes of the search buffer and the look-ahead buffer. With a finite size for the search buffer, the LZ77 algorithms will not work well in the case where repeated patterns are apart from each other by a distance longer than the size of the search buffer. With a finite size for the sliding window, the LZ77 algorithms will not work well in the case where matching strings are longer than the window. In order to be efficient, however, these sizes cannot be very large.

In order to overcome the problem, the LZ78 algorithms work in a different way. They do not use the sliding window at all. Instead of using the most recently encoded portion of the input text as a dictionary, the LZ78 algorithms use the index of the longest matched string as an entry of the dictionary. That is, each matched string cascaded with its immediate next symbol is compared with the existing entries of the dictionary. If this combination (a new string) does not find a match in the dictionary constructed at the moment, the combination will be included as an entry in the dictionary. Otherwise, the next symbol in the input text will be appended to the combination and the enlarged new combination will be checked with the dictionary. The process continues until the new combination cannot find a match in the dictionary. Among the several variants of the LZ78 algorithms, the LZW algorithm is perhaps the most important one. It only needs to encode the indexes of the longest matched strings to the dictionary. It can be shown that the decoder can decode the input text stream from the given index stream. In doing so, the same dictionary as that established in the encoder needs to be reconstructed at the decoder, and this can be implemented since the same rule used in the encoding is known in the decoder.

The size of the dictionary cannot be infinitely large because, as mentioned above, the coding efficiency will not be high. The common practice of the LZ78 algorithms is to keep the dictionary fixed once a certain size has been reached and the performance of the encoding is satisfactory. Otherwise, the dictionary will be set to empty and will be reconstructed from scratch.

Considering the fact that there are several international standards concerning still image coding (for both bilevel and multilevel images), a brief summary of them and a performance comparison have been presented in this chapter. At the beginning of this chapter, a description of the discrete Markov source and its nth extensions was provided. The Markov source and the autoregressive model serve as important models for the dependent information sources.

6.7 EXERCISES

6-1. Draw the state diagram of a second-order Markov source with two symbols in the source alphabet. That is, $S = \{s_1, s_2\}$. It is assumed that the conditional probabilities are

$$p(s_1 | s_1 s_1) = p(s_2 | s_2 s_2) = 0.7,$$

$$p(s_2 | s_1 s_1) = p(s_1 | s_2 s_2) = 0.3, \text{ and}$$

$$p(s_1 | s_1 s_2) = p(s_1 | s_2 s_1) = p(s_2 | s_1 s_2) = p(s_2 | s_2 s_1) = 0.5.$$

6-2. What are the definitions of raster algorithm and area algorithm in binary image coding? To which category does 1-D RLC belong? To which category does 2-D RLC belong?

6-3. What effect does a transmission error have on 1-D RLC and 2-D RLC, respectively? What is the function of the codeword EOL?

6-4. Make a convincing argument that the "modified" Huffman (MH) algorithm reduces the requirement of large storage space.

6-5. Which three different modes does 2-D RLC have? How do you view the vertical mode?

6-6. Using your own words, describe the encoding and decoding processes of the LZ77 algorithms. Go through Example 6.2.

6-7. Using your own words, describe the encoding and decoding processes of the LZW algorithm. Go through Example 6.3.

6-8. Read the reference paper (Arps and Truong, 1994), which is an excellent survey on the international standards for lossless still image compression. Pay particular attention to all the figures and to Table 1.

REFERENCES

Abramson, N. *Information Theory and Coding*, New York: McGraw-Hill, 1963.

Arps, R. B. Binary Image Compression, in *Image Transmission Techniques*, W. K. Pratt (Ed.), New York: Academic Press, 1979.

Arps, R. B. and T. K. Truong, Comparison of international standards for lossless still image compression, *Proc. IEEE,* 82(6), 889-899, 1994.

Bell, T. C., J. G. Cleary, and I. H. Witten, *Text Compression*, Englewood Cliffs, NJ: Prentice-Hall, 1990.

Gonzalez, R. C. and R. E. Woods, *Digital Image Processing*, Reading, MA: Addison-Wesley, 1992.

Hunter, R. and A. H. Robinson, International digital facsimile coding standards, *Proc. IEEE,* 68(7), 854-867, 1980.

Laemmel, A. E. Coding Processes for Bandwidth Reduction in Picture Transmission, Rep. R-246-51, PIB-187, Microwave Res. Inst., Polytechnic Institute of Brooklyn, New York.

Nelson, M. and J.-L. Gailly, *The Data Compression Book*, 2nd ed., New York: M&T Books, 1995.

Sayood, K. *Introduction to Data Compression*, San Francisco, CA: Morgan Kaufmann Publishers, 1996.

Shannon, C. E. and W. Weaver, *The Mathematical Theory of Communication*, Urbana, IL: University of Illinois Press, 1949.

Welch, T. A technique for high-performance data compression, *IEEE Trans. Comput.*, 17(6), 8-19, 1984.

Ziv, J. and A. Lempel, A universal algorithm for sequential data compression, *IEEE Trans. Inf. Theory,* 23(3), 337-343, 1977.

Ziv, J. and A. Lempel, Compression of individual sequences via variable-rate coding, *IEEE Trans. Inf. Theory,* 24(5), 530-536, 1978.

IPR2021-00827
Unified EX1008 Page 179

IPR2021-00827
Unified EX1008 Page 180

Section II

Still Image Compression

IPR2021-00827
Unified EX1008 Page 181

IPR2021-00827
Unified EX1008 Page 182

7 Still Image Coding Standard: JPEG

In this chapter, the JPEG standard is introduced. This standard allows for lossy and lossless encoding of still images and four distinct modes of operation are supported: sequential DCT-based mode, progressive DCT-based mode, lossless mode and hierarchical mode.

7.1 INTRODUCTION

Still image coding is an important application of data compression. When an analog image or picture is digitized, each pixel is represented by a fixed number of bits, which correspond to a certain number of gray levels. In this uncompressed format, the digitized image requires a large number of bits to be stored or transmitted. As a result, compression become necessary due to the limited communication bandwidth or storage size. Since the mid-1980s, the ITU and ISO have been working together to develop a joint international standard for the compression of still images. Officially, JPEG [jpeg] is the ISO/IEC international standard 10918-1; digital compression and coding of continuous-tone still images, or the ITU-T Recommendation T.81. JPEG became an international standard in 1992. The JPEG standard allows for both lossy and lossless encoding of still images. The algorithm for lossy coding is a DCT-based coding scheme. This is the baseline of JPEG and is sufficient for many applications. However, to meet the needs of applications that cannot tolerate loss, e.g., compression of medical images, a lossless coding scheme is also provided and is based on a predictive coding scheme. From the algorithmic point of view, JPEG includes four distinct modes of operation, namely, sequential DCT-based mode, progressive DCT-based mode, lossless mode, and hierarchical mode. In the following sections, an overview of these modes is provided. Further technical details can be found in the books by Pennelbaker and Mitchell (1992) and Symes (1998).

In the sequential DCT-based mode, an image is first partitioned into blocks of 8×8 pixels. The blocks are processed from left to right and top to bottom. The 8×8 two-dimensional Forward DCT is applied to each block and the 8×8 DCT coefficients are quantized. Finally, the quantized DCT coefficients are entropy encoded and output as part of the compressed image data.

In the progressive DCT-based mode, the process of block partitioning and Forward DCT transform is the same as in the sequential DCT-based mode. However, in the progressive mode, the quantized DCT coefficients are first stored in a buffer before the encoding is performed. The DCT coefficients in the buffer are then encoded by a multiple scanning process. In each scan, the quantized DCT coefficients are partially encoded by either spectral selection or successive approximation. In the method of spectral selection, the quantized DCT coefficients are divided into multiple spectral bands according to a zigzag order. In each scan, a specified band is encoded. In the method of successive approximation, a specified number of most significant bits of the quantized coefficients are first encoded and the least significant bits are then encoded in subsequent scans.

The difference between sequential coding and progressive coding is shown in Figure 7.1. In the sequential coding an image is encoded part by part according to the scanning order, while in the progressive coding the image is encoded by a multiscanning process and in each scan the full image is encoded to a certain quality level.

As mentioned earlier, lossless coding is achieved by a predictive coding scheme. In this scheme, three neighboring pixels are used to predict the current pixel to be coded. The prediction difference

IPR2021-00827
Unified EX1008 Page 183

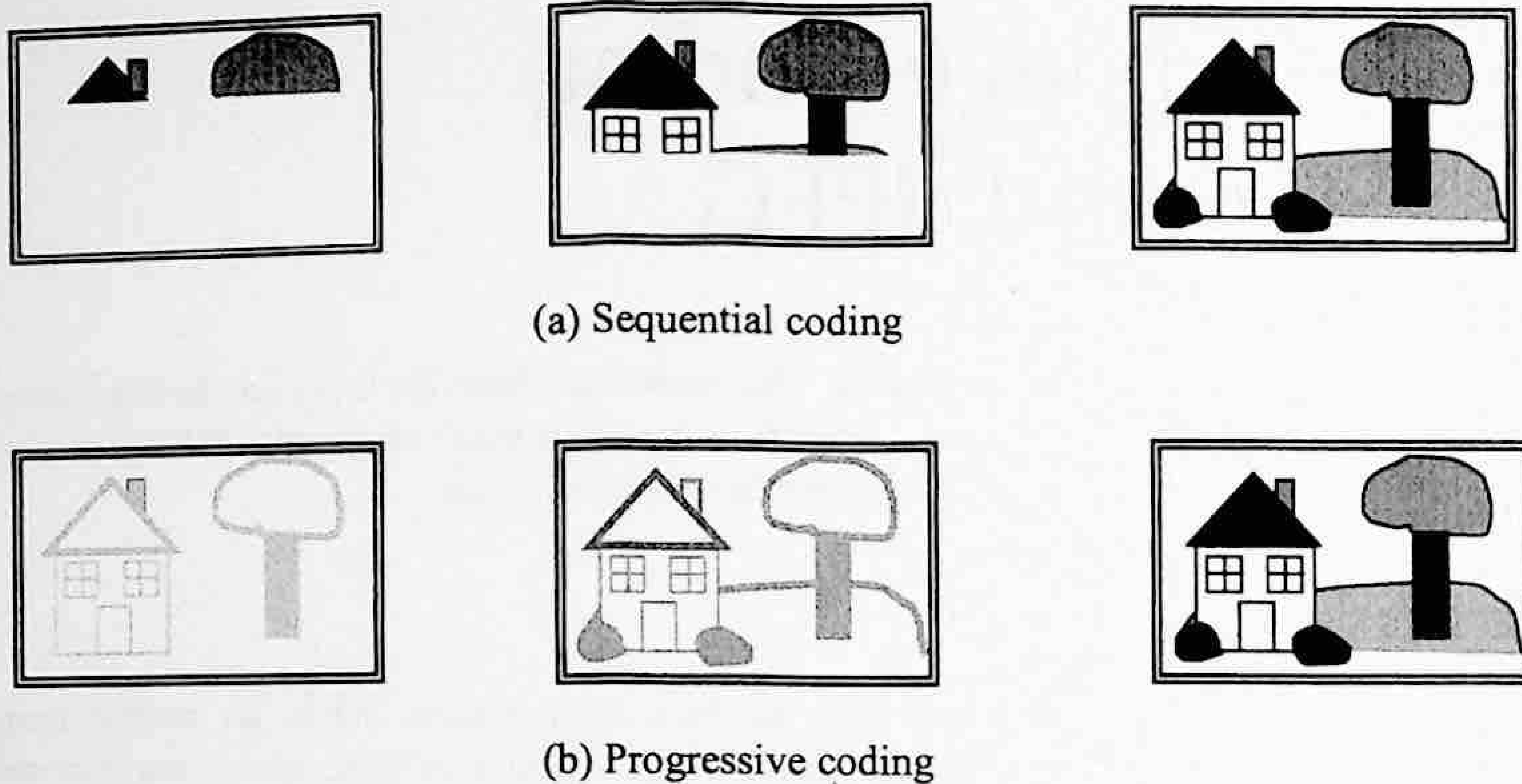


FIGURE 7.1 (a) Sequential coding, (b) progressive coding.

is entropy coded using either Huffman or arithmetic coding. Since the prediction is not quantized, the coding is lossless.

Finally, in the hierarchical mode, an image is first spatially down-sampled to a multilayered pyramid, resulting in a sequence of frames as shown in Figure 7.2. This sequence of frames is encoded by a predictive coding scheme. Except for the first frame, the predictive coding process is applied to the differential frames, i.e., the differences between the frame to be coded and the predictive reference frame. It is important to note that the reference frame is equivalent to the previous frame that would be reconstructed in the decoder. The coding method for the difference frame may use the DCT-based coding method, the lossless coding method, or the DCT-based processes with a final lossless process. Down-sampling and up-sampling filters are used in the hierarchical mode. The hierarchical coding mode provides a progressive presentation similar to the progressive DCT-based mode, but is also useful in the applications that have multiresolution requirements. The hierarchical coding mode also provides the capability of progressive coding to a final lossless stage.

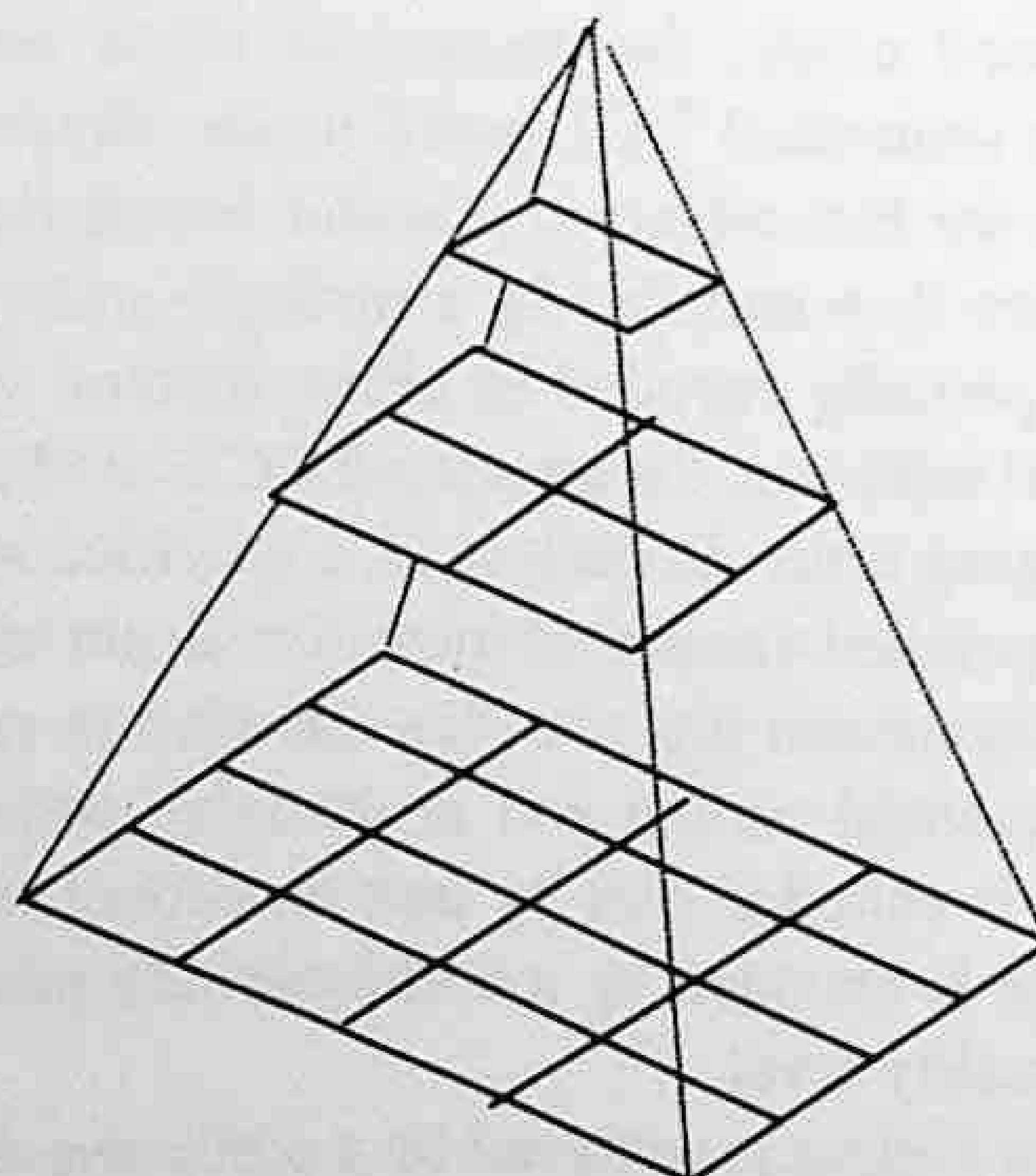


FIGURE 7.2 Hierarchical multiresolution encoding.

IPR2021-00827
Unified EX1008 Page 184

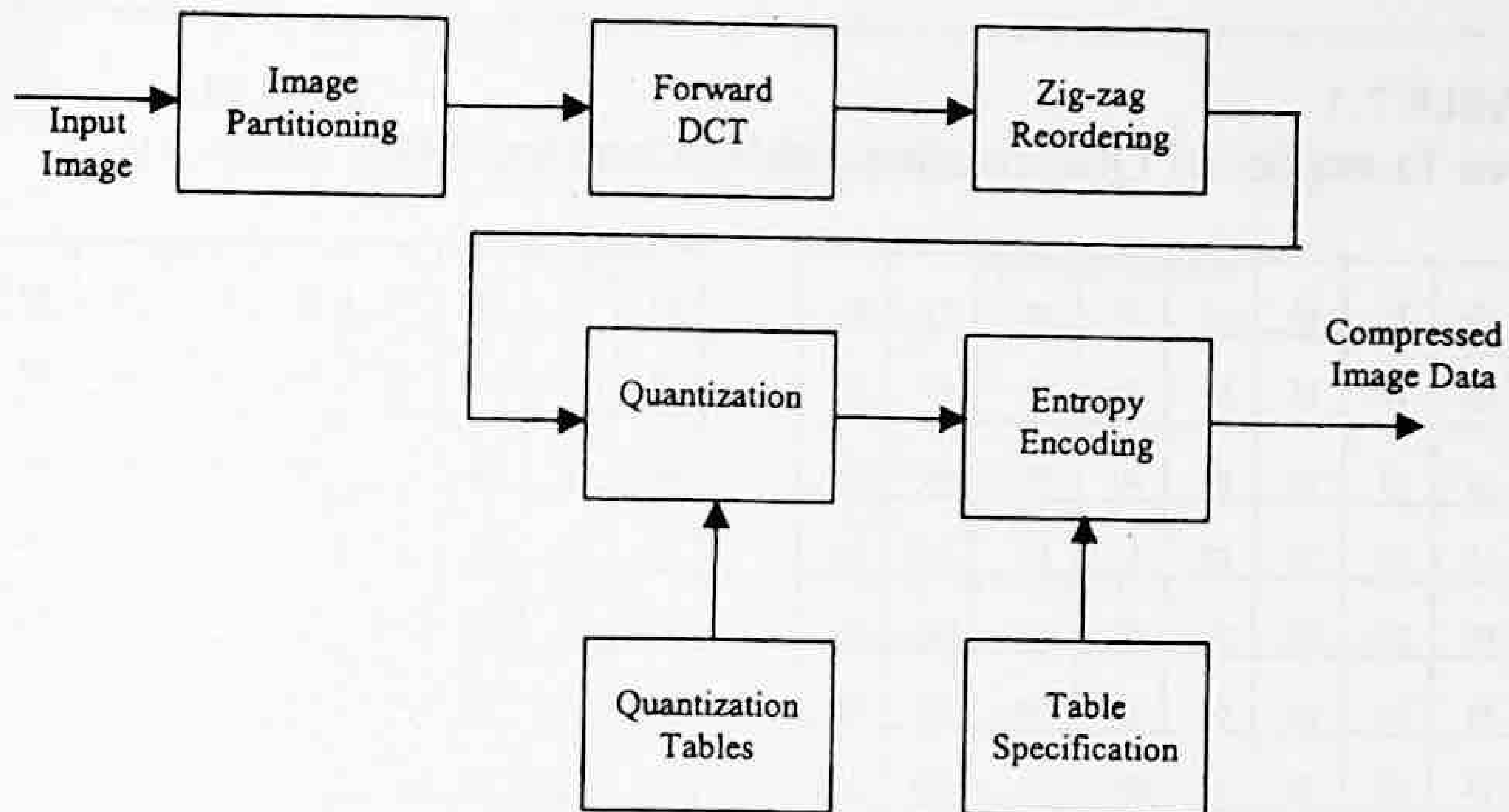


FIGURE 7.3 Block diagram of a sequential DCT-based encoding process.

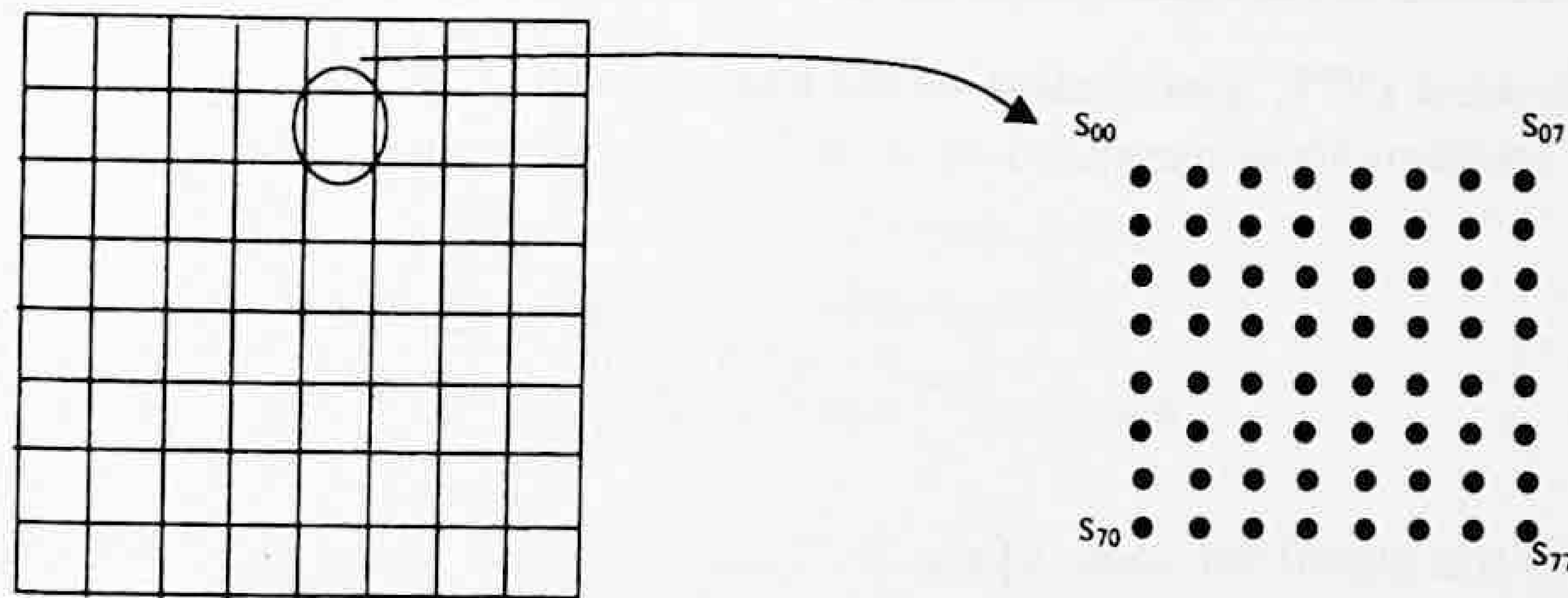


FIGURE 7.4 Partitioning to 8 × 8 blocks.

7.2 SEQUENTIAL DCT-BASED ENCODING ALGORITHM

The sequential DCT-based coding algorithm is the baseline algorithm of the JPEG coding standard. A block diagram of the encoding process is shown in Figure 7.3. As shown in Figure 7.4, the digitized image data are first partitioned into blocks of 8 × 8 pixels. The two-dimensional forward DCT is applied to each 8 × 8 block. The two-dimensional forward and inverse DCT of 8 × 8 block are defined as follows:

$$\text{FDCT:} \quad S_{uv} = \frac{1}{4} C_u C_v \sum_{i=0}^{7} \sum_{j=0}^{7} s_{ij} \cos\frac{(2i+1)u\pi}{16} \cos\frac{(2j+1)v\pi}{16} .$$

$$\text{IDCT:} \quad s_{ij} = \frac{1}{4} \sum_{u=0}^{7} \sum_{v=0}^{7} C_u C_v S_{uv} \cos\frac{(2i+1)u\pi}{16} \cos\frac{(2j+1)v\pi}{16} \tag{7.1}$$

$$C_u C_v = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } u, v = 0 \\ 1 & \text{otherwise} \end{cases}$$

where s_{ij} is the value of the pixel at position (i,j) in the block, and S_{uv} is the transformed (u,v) DCT coefficient.

TABLE 7.1
Two Examples of Quantization Tables Used by JPEG

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Luminance quantization table

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

Chrominance quantization table

After the forward DCT, quantization of the transformed DCT coefficients is performed. Each of the 64 DCT coefficients is quantized by a uniform quantizer:

$$S_{quv} = round\left(\frac{S_{uv}}{Q_{uv}}\right) \tag{7.2}$$

where the S_{quv} is the quantized value of the DCT coefficient, S_{uv}, and Q_{uv} is the quantization step obtained from the quantization table. There are four quantization tables that may be used by the encoder, but there is no default quantization table specified by the standard. Two particular quantization tables are shown in Table 7.1.

At the decoder, the dequantization is performed as follows:

$$R_{quv} = S_{quv} \times Q_{uv} \tag{7.3}$$

where R_{quv} is the value of the dequantized DCT coefficient. After quantization, the DC coefficient, S_{q00}, is treated separately from the other 63 AC coefficients. The DC coefficients are encoded by a predictive coding scheme. The encoded value is the difference (*DIFF*) between the quantized DC coefficient of the current block (S_{q00}) and that of the previous block of the same component (*PRED*):

$$DIFF = S_{q00} - PRED \tag{7.4}$$

The value of *DIFF* is entropy coded with Huffman tables. More specifically, the two's complement of the possible *DIFF* magnitudes are grouped into 12 categories, "SSSS". The Huffman codes for these 12 difference categories and additional bits are shown in the Table 7.2.

For each nonzero category, additional bits are added to the codeword to uniquely identify which difference within the category actually occurred. The number of additional bits is defined by "SSSS" and the additional bits are appended to the least significant bit of the Huffman code (most significant bit first) according to the following rule. If the difference value is positive, the "SSSS" low-order bits of *DIFF* are appended; if the difference value is negative, then the "SSSS" low-order bits of *DIFF-1* are appended. As an example, the Huffman tables used for coding the luminance and chrominance DC coefficients are shown in Tables 7.3 and 7.4, respectively. These two tables have been developed from the average statistics of a large set of images with 8-bit precision.

IPR2021-00827
Unified EX1008 Page 186

TABLE 7.2
Huffman Coding of DC Coefficients

SSSS	DIFF Values	Additional Bits
0	0	–
1	–1,1	0,1
2	–3,–2,2,3	00,01,10,11
3	–7,…,–4,4,…,7	000,…,011,100,..111
4	–15,…,–8,8,…,15	0000,..0111,1000,…,1111
5	–31,…,–16,16,…,31	00000,…,01111,10000,…,11111
6	–63,…–32,32,…63	…,…
7	–127,…,–64,64,…,127	…,…
8	–255,…,–128,128,…,255	…,…
9	–511,…,–256,256,…,511	…,…
10	–1023,…,–512,512,…,1023	…,…
11	–2047,…,–1024,1024,…,2047	…,…

TABLE 7.3
Huffman 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

In contrast to the coding of DC coefficients, the quantized AC coefficients are arranged to a zigzag order before being entropy coded. This scan order is shown in Figure 7.5.

According to the zigzag scanning order, the quantized coefficients can be represented as:

$$ZZ(0) = S_{q00},\ ZZ(1) = S_{q01},\ ZZ(2) = S_{q10},\ \ldots,\ ZZ(63) = S_{q77}. \tag{7.5}$$

Since many of the quantized AC coefficients become zero, they can be very efficiently encoded by exploiting the run of zeros. The run-length of zeros are identified by the nonzero coefficients. An 8-bit code 'RRRRSSSS' is used to represent the nonzero coefficient. The four least significant bits, 'SSSS', define a category for the value of the next nonzero coefficient in the zigzag sequence, which ends the zero run. The four most significant bits, 'RRRR', define the run-length of zeros in the zigzag sequence or the position of the nonzero coefficient in the zigzag sequence. The composite value, RRRRSSSS, is shown in Figure 7.6. The value 'RRRRSSSS' = '11110000' is defined as ZRL, "RRRR" = "1111" represents a run-length of 16 zeros and "SSSS" = "0000" represents a zero amplitude. Therefore, ZRL is used to represent a run-length of 16 zero coefficients followed

IPR2021-00827
Unified EX1008 Page 187

TABLE 7.4
Huffman table for chrominance DC coefficient differences

Category	Code Length	Codeword
0	2	00
1	2	01
2	2	10
3	3	110
4	4	1110
5	5	11110
6	6	111110
7	7	1111110
8	8	11111110
9	9	111111110
10	10	1111111110
11	11	11111111110

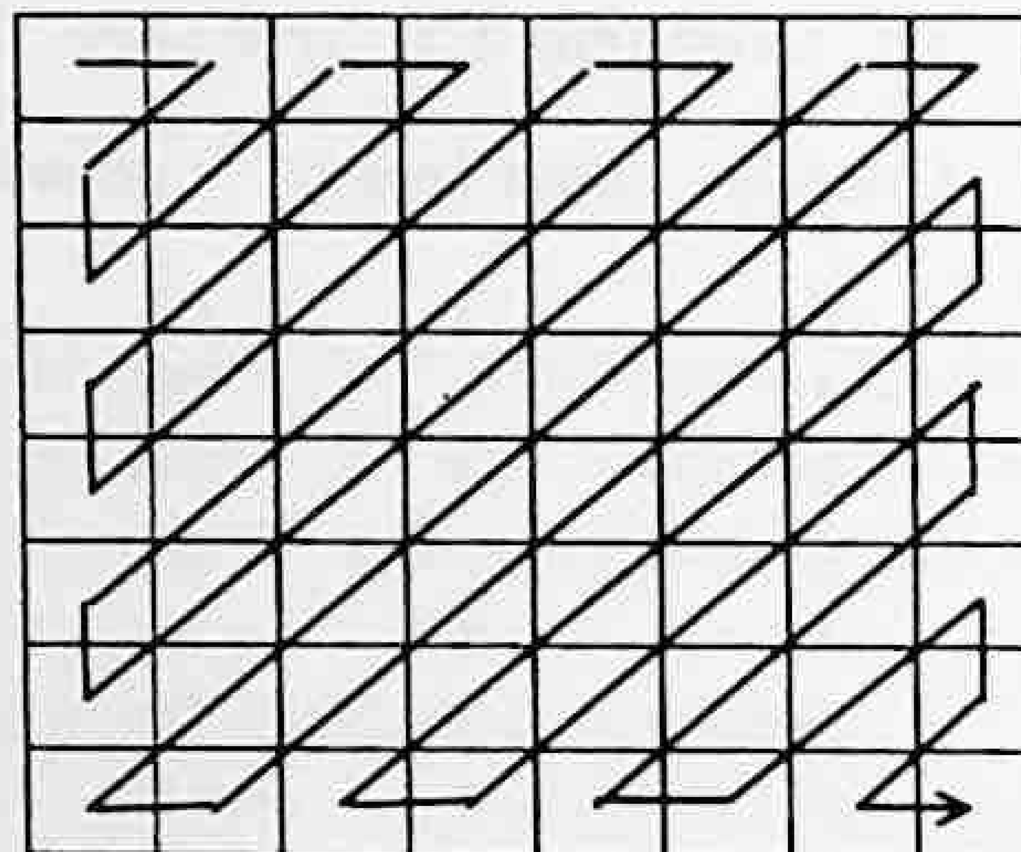


FIGURE 7.5 Zigzag scanning order of DCT coefficients.

FIGURE 7.6 Two-dimensional value array for Huffman coding.

by a zero-amplitude coefficient, it is not an *abbreviation*. In the case of a run-length of zero coefficients that exceeds 15, multiple symbols will be used. A special value 'RRRRSSSS' = '00000000' is used to code the end-of-block (EOB). An EOB occurs when the remaining coefficients in the block are zeros. The entries marked "N/A" are undefined.

TABLE 7.5
Huffman Coding for AC Coefficients

Category (SSSS)	AC Coefficient Range
1	–1,1
2	–3,–2,2,3
3	–7,…,–4,4,…,7
4	–15,…,–8,8,…,15
5	–31,…,–16,16,…,31
6	–63,…,–32,32,…,63
7	–127,…,–64,.64,…,127
8	–255,…,–128,128,…,255
9	–511,…,–256,256,…,511
10	–1023,.,–512,512,…,1023
11	–2047,…,–1024,1024,…,2047

The composite value, RRRRSSSS, is then Huffman coded. SSSS is actually the number to indicate "category" in the Huffman code table. The coefficient values for each category are shown in Table 7.5.

Each Huffman code is followed by additional bits that specify the sign and exact amplitude of the coefficients. As with the DC code tables, the AC code tables have also been developed from the average statistics of a large set of images with 8-bit precision. Each composite value is represented by a Huffman code in the AC code table. The format for the additional bits is the same as in the coding of DC coefficients. The value of SSSS gives the number of additional bits required to specify the sign and precise amplitude of the coefficient. The additional bits are either the low-order SSSS bits of ZZ(k) when ZZ(k) is positive, or the low-order SSSS bits of ZZ(k)-1 when ZZ(k) is negative. Here, ZZ(k) is the *kth* coefficient in the zigzag scanning order of coefficients being coded. The Huffman tables for AC coefficients can be found in Annex K of the JPEG standard (jpeg) and are not listed here due to space limitations.

As described above, Huffman coding is used as the means of entropy coding. However, an adaptive arithmetic coding procedure can also be used. As with the Huffman coding technique, the binary arithmetic coding technique is also lossless. It is possible to transcode between two systems without either of the FDCT or IDCT processes. Since this transcoding is a lossless process, it does not affect the picture quality of the reconstructed image. The arithmetic encoder encodes a series of binary symbols, zeros or ones, where each symbol represents the possible result of a binary decision. The binary decisions include the choice between positive and negative signs, a magnitude being zero or nonzero, or a particular bit in a sequence of binary digits being zero or one. There are four steps in the arithmetic coding: initializing the statistical area, initializing the encoder, terminating the code string, and adding restart markers.

7.3 PROGRESSIVE DCT-BASED ENCODING ALGORITHM

In progressive DCT-based coding, the input image is first partitioned to blocks of 8×8 pixels. The two-dimensional 8×8 DCT is then applied to each block. The transformed DCT-coefficient data are then encoded with multiple scans. At each scan, a portion of the transformed DCT coefficient data is encoded. This partially encoded data can be reconstructed to obtain a full image size with lower picture quality. The coded data of each additional scan will enhance the reconstructed image quality until the full quality has been achieved at the completion of all scans. Two methods have been used in the JPEG standard to perform the DCT-based progressive coding. These include spectral selection and successive approximation.

IPR2021-00827
Unified EX1008 Page 189

In the method of spectral selection, the transformed DCT coefficients are first reordered as a zigzag sequence and then divided into several bands. A frequency band is defined in the scan header by specifying the starting and ending indexes in the zigzag sequence. The band containing the DC coefficient is encoded at the first scan. In the following scan, it is not necessary for the coding procedure to follow the zigzag ordering.

In the method of the successive approximation, the DCT coefficients are first reduced in precision by the point transform. The point transform of the DCT coefficients is an arithmetic shift right by a specified number of bits, or division by a power of 2 (near zero, there is slight difference in truncation of precision between an arithmetic shift and division by 2, see annex K10 of [jpeg]). This specified number is the successive approximation of bit position. To encode using successive approximations, the significant bits of the DCT coefficient are encoded in the first scan, and each successive scan that follows progressively improves the precision of the coefficient by one bit. This continues until full precision is reached.

The principles of spectral selection and successive approximation are shown in Figure 7.7. For both methods, the quantized coefficients are coded with either Huffman or arithmetic codes at each scan. In spectral selection and the first scan of successive approximation for an image, the AC coefficient coding model is similar to that used in the sequential DCT-based coding mode. However, the Huffman code tables are extended to include coding of runs of end-of-bands (EOBs). For distinguishing the end-of-band and end-of-block, a number, n, which is used to indicate the range of run length, is added to the end-of-band (EOBn). The EOBn code sequence is defined as follows. Each EOBn is followed by an extension field, which has the minimum number of bits required to specify the run length. The end-of-band run structure allows efficient coding of blocks which have only zero coefficients. For example, an EOB run of length 5 means that the current block and the next 4 blocks have an end-of-band with no intervening nonzero coefficients. The Huffman coding structure of the subsequent scans of successive approximation for a given image is similar to the coding structure of the first scan of that image. Each nonzero quantized coefficient is described by a composite 8-bit run length-magnitude value of the form: RRRRSSSS. The four most significant bits, RRRR, indicate the number of zero coefficients between the current coefficient and the previously coded coefficient. The four least significant bits, SSSS, give the magnitude category of the nonzero coefficient. The run length-magnitude composite value is Huffman coded. Each Huffman code is followed by additional bits: one bit is used to code the sign of the nonzero coefficient and another bit is used to code the correction, where "0" means no correction and "1" means add one to the decoded magnitude of the coefficient. Although the above technique has been described using Huffman coding, it should be noted that arithmetic encoding can also be used in its place.

7.4 LOSSLESS CODING MODE

In the lossless coding mode, the coding method is spatially based coding instead of DCT-based coding. However, the coding method is extended from the method for coding the DC coefficients in the sequential DCT-based coding mode. Each pixel is coded with a predictive coding method, where the predicted value is obtained from one of three one-dimensional or one of four two-dimensional predictors, which are shown in Figure 7.8.

In Figure 7.8, the pixel to be coded is denoted by x, and the three causal neighbors are denoted by a, b, and c. The predictive value of x, Px, is obtained from three neighbors, a, b, and c in the one of seven ways as listed in Table 7.6.

In Table 7.6, the selection value 0 is only used for differential coding in the hierarchical coding mode. Selections 1, 2, and 3 are one-dimensional predictions and 4, 5, 6, and 7 are two-dimensional predictions. Each prediction is performed with full integer precision, and without clamping of either the underflow or overflow beyond the input bounds. In order to achieve lossless coding, the prediction differences are coded with either Huffman coding or arithmetic coding. The prediction

IPR2021-00827
Unified EX1008 Page 190

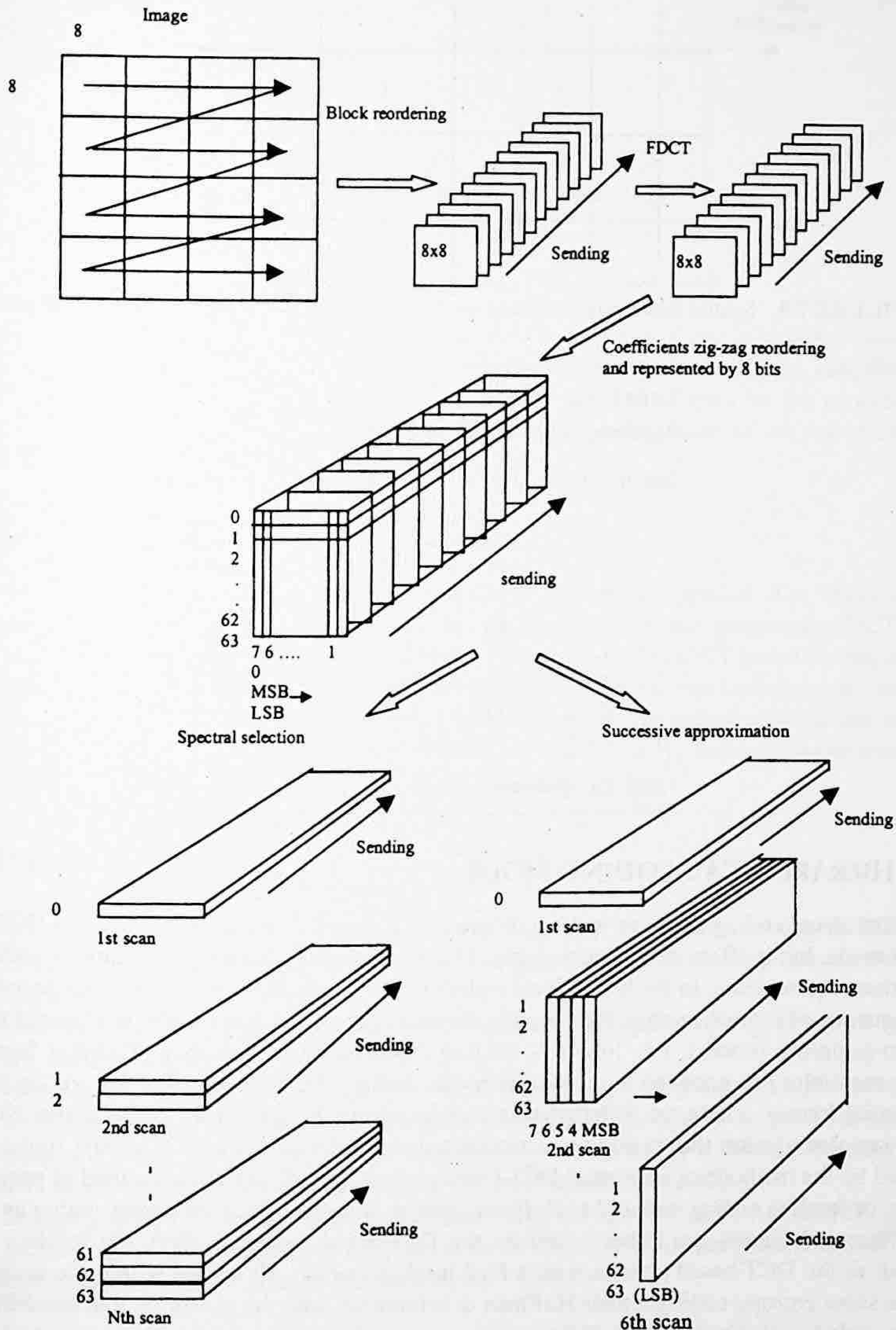


FIGURE 7.7 Progressive coding with spectral selection and successive approximation.

difference values can be from 0 to 2^{16} for 8-bit pixels. The Huffman tables developed for coding DC coefficients in the sequential DCT-based coding mode are used with one additional entry to code the prediction differences. For arithmetic coding, the statistical model defined for the DC coefficients in the sequential DCT-based coding mode is generalized to a two-dimensional form in which differences are conditioned on the pixel to the left and the line above.

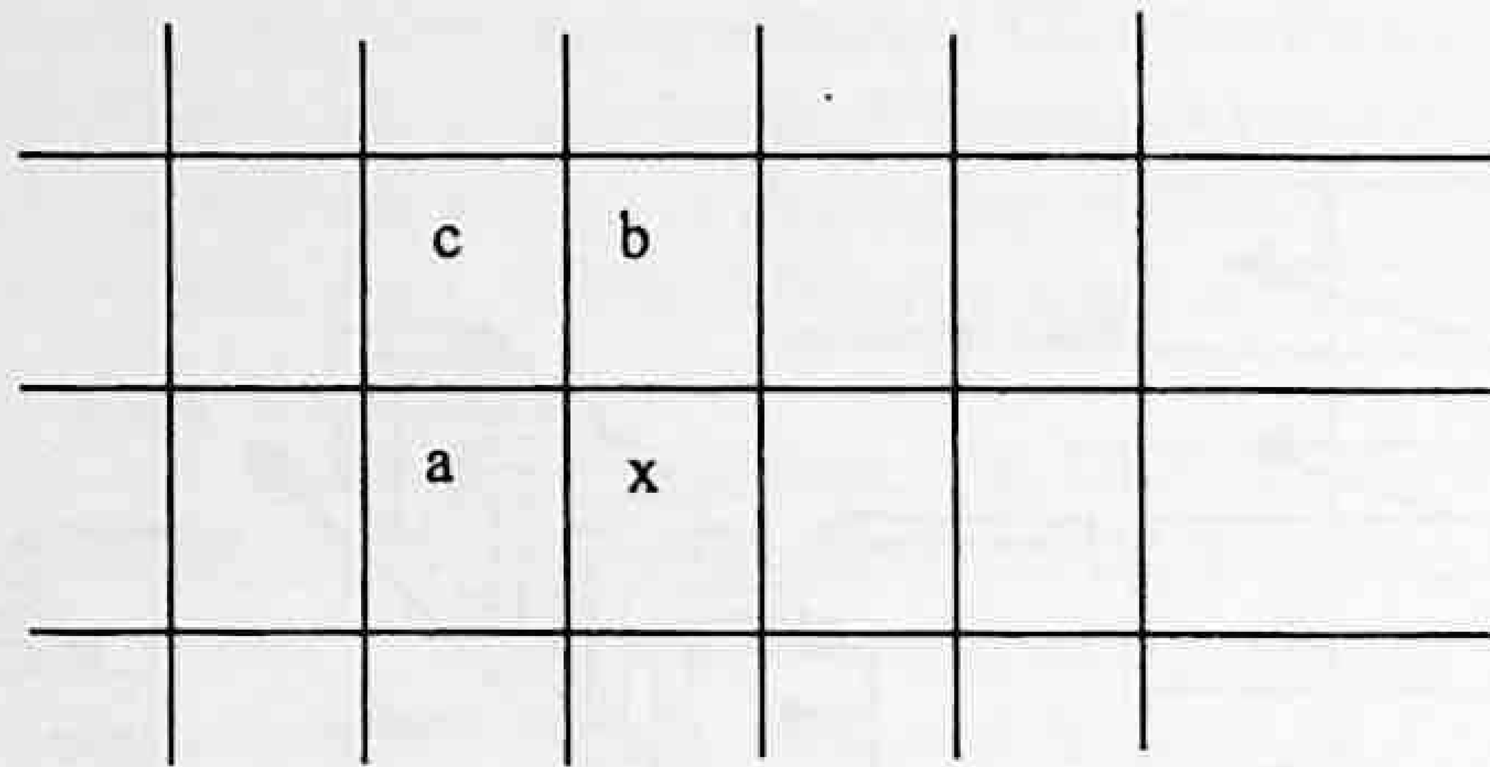


FIGURE 7.8 Spatial relationship between the pixel to be coded and three decoded neighbors.

TABLE 7.6
Predictors for Lossless Coding

Selection-Value	Prediction
0	No prediction (hierarchical mode)
1	Px = a
2	Px = b
3	Px = c
4	Px = a+b-c
5	Px = a + ((b-c)/2)[a]
6	Px = b + ((a-c)/2)[a]
7	Px = (a+b)/2

[a] Shift right arithmetic operation.

7.5 HIERARCHICAL CODING MODE

The hierarchical coding mode provides a progressive coding similar to the progressive DCT-based coding mode, but it offers more functionality. This functionality addresses applications with multi-resolution requirements. In the hierarchical coding mode, an input image frame is first decomposed to a sequence of frames, such as the pyramid shown in Figure 7.2. Each frame is obtained through a down-sampling process, i.e., low-pass filtering followed by subsampling. The first frame (the lowest resolution) is encoded as a nondifferential frame. The following frames are encoded as differential frames, where the differential is with respect to the previously coded frame. Note that an up-sampled version that would be reconstructed in the decoder is used. The first frame can be encoded by the methods of sequential DCT-based coding, spectral selection, method of progressive coding, or lossless coding with either Huffman code or arithmetic code. However, within an image, the differential frames are either coded by the DCT-based coding method, the lossless coding method, or the DCT-based process with a final lossless coding. All frames within the image must use the same entropy coding, either Huffman or arithmetic, with the exception that nondifferential frames coded with the baseline coding may occur in the same image with frames coded with arithmetic coding methods. The differential frames are coded with the same method used for the nondifferential frames except the final frame. The final differential frame for each image may use a differential lossless coding method.

In the hierarchical coding mode, resolution changes in frames may occur. These resolution changes occur if down-sampling filters are used to reduce the spatial resolution of some or all frames of an image. When the resolution of a reference frame does not match the resolution of the frame to be coded, a up-sampling filter is used to increase the resolution of the reference frame. The block diagram of coding of a differential frame is shown in Figure 7.9.

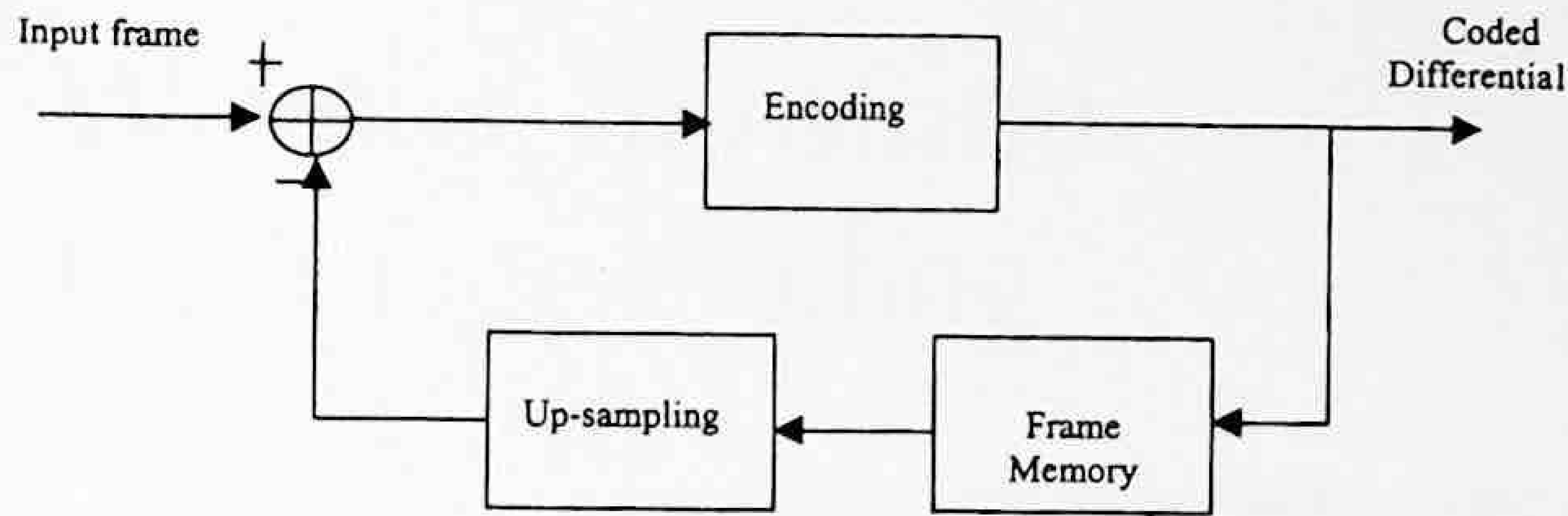


FIGURE 7.9 Hierarchical coding of a differential frame.

The up-sampling filter increases the spatial resolution by a factor of two in both horizontal and vertical directions by using bilinear interpolation of two neighboring pixels. The up-sampling with bilinear interpolation is consistent with the down-sampling filter that is used for the generation of down-sampled frames. It should be noted that the hierarchical coding mode allows one to improve the quality of the reconstructed frames at a given spatial resolution.

7.6 SUMMARY

In this chapter, the still image coding standard, JPEG, has been introduced. The JPEG coding standard includes four coding modes: sequential DCT-based coding mode, progressive DCT-based coding mode, lossless coding mode, and hierarchical coding mode. The DCT-based coding method is probably the one that most of us are familiar with; however, the lossless coding modes in JPEG which use a spatial domain predictive coding process have many interesting applications as well. For each coding mode, entropy coding can be implemented with either Huffman coding or arithmetic coding. JPEG has been widely adopted for many applications.

7.7 EXERCISES

7-1. What is the difference between sequential coding and progressive coding in JPEG? Conduct a project to encode an image with sequence coding and progressive coding, respectively.

7-2. Use the JPEG lossless mode to code several images and explain why different bit rates are obtained.

7-3. Generate a Huffman code table using a set of images with 8-bit precision (aproximately 2~3) using the method presented in Annex C of the JPEG specification. This set of images is called the training set. Use this table to code an image within the training set and an image which is not in the training set, and explain the results.

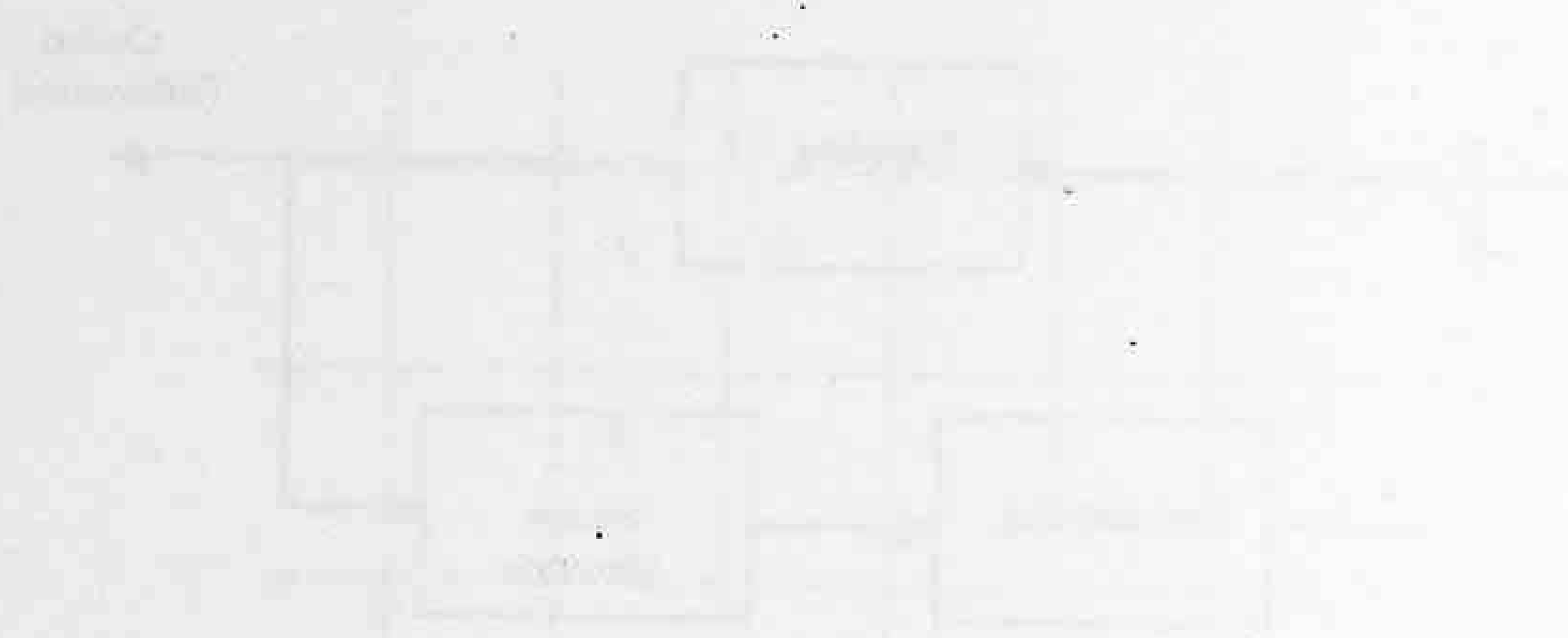
7-4. Design a three-layer progressive JPEG coder using (a) spectral selection, and (b) progressive approximation (0.3 bits per pixel at the first layer, 0.2 bits per pixel at the second layer, and 0.1 bits per pixel at the third layer).

REFERENCES

Digital compression and coding of continuous-tone still images. Requirements and Guidelines, ISO-/IEC International Standard 10918-1, CCITT T.81, September, 1992.

Pennelbaker, W. B. and J. L. Mitchell, *JPEG: Still Image Data Compression Standard*, Van Nostrand Reinhold, New York, 1992.

Symes, P. *Compression: Fundamental Compression Techniques and an Overview of the JPEG and MPEG Compression Systems*, McGraw-Hill, New York, 1998.



8 Wavelet Transform for Image Coding

During the last decade, a number of signal processing applications have emerged using wavelet theory. Among those applications, the most widespread developments have occurred in the area of data compression. Wavelet techniques have demonstrated the ability to provide not only high coding efficiency, but also spatial and quality scalability features. In this chapter, we focus on the utility of the wavelet transform for image data compression applications.

8.1 REVIEW OF THE WAVELET TRANSFORM

8.1.1 DEFINITION AND COMPARISON WITH SHORT-TIME FOURIER TRANSFORM

The wavelet transform, as a specialized research field, started over a decade ago (Grossman and Morlet, 1984). To better understand the theory of wavelets, we first give a very short review of the Short-Time Fourier Transform (STFT) since there are some similarities between the STFT and the wavelet transform. As we know, the STFT uses sinusoidal waves as its orthogonal basis and is defined as:

$$F(\omega,\tau) = \int_{-\infty}^{+\infty} f(t)w(t-\tau)e^{-j\omega t}dt \tag{8.1}$$

where $w(t)$ is a time-domain windowing function, the simplest of which is a rectangular window that has a unit value over a time interval and has zero elsewhere. The value τ is the starting position of the window. Thus, the STFT maps a function $f(t)$ into a two-dimensional plane (ω,τ). The STFT is also referred to as Gabor transform (Cohen, 1989). Similar to the STFT, the wavelet transform also maps a time or spatial function into a two-dimensional function in a and τ (ω and τ for STFT). The wavelet transform is defined as follows. Let $f(t)$ be any square integrable function, i.e., it satisfies:

$$\int_{-\infty}^{+\infty} |f(t)|^2 dt < \infty \tag{8.2}$$

The continuous-time wavelet transform of $f(t)$ with respect to a wavelet $\psi(t)$ is defined as:

$$W(a,\tau) = \int_{-\infty}^{+\infty} f(t)\frac{1}{\sqrt{|a|}}\psi^*\left(\frac{t-\tau}{a}\right)dt \tag{8.3}$$

where a and τ are real variables and * denotes complex conjugation. The wavelet is defined as:

$$\psi_{a\tau}(t) = |a|^{-1/2}\psi\left(\frac{t-\tau}{a}\right) \tag{8.4}$$

IPR2021-00827
Unified EX1008 Page 195

The above equation represents a set of functions that are generated from a single function, $\psi(t)$, by dilations and translations. The variable τ represents the time shift and the variable a corresponds to the amount of time-scaling or dilation. If $a > 1$, there is an expansion of $\psi(t)$, while if $0 < a < 1$, there is a contraction of $\psi(t)$. For negative values of a, the wavelet experiences a time reversal in combination with a dilation. The function, $\psi(t)$, is referred to as the mother wavelet and it must satisfy two conditions:

1. The function integrates to zero:

$$\int_{-\infty}^{+\infty} \psi(t)dt = 0 \tag{8.5}$$

2. The function is square integrable, or has finite energy:

$$\int_{-\infty}^{+\infty} \psi|(t)|^2 dt < \infty \tag{8. 6}$$

The continuous-time wavelet transform can now be rewritten as:

$$W(a,\tau) = \int_{-\infty}^{+\infty} f(t)\psi_{a\tau}^{*}(t)dt \tag{8.7}$$

In the following, we give two well-known examples of $\psi(t)$ and their Fourier transforms. The first example is the Morlet (modulated Gaussian) wavelet (Daubechies, 1990),

$$\Psi(\omega) = \sqrt{2\pi}\, e^{\frac{(\omega-\omega_0)^2}{2}} \tag{8.8}$$

and the second example is the Haar wavelet:

$$\psi = \begin{cases} 1 & 0 \le t \le 1/2 \\ -1 & 1/2 \le t \le 1 \\ 0 & otherwise \end{cases} \tag{8.9}$$

$$\psi(\omega) = je^{-j\frac{\omega}{2}} \frac{sm^2(\omega/4)}{\omega/4}$$

From the above definition and examples, we can find that the wavelets have zero DC value. This is clear from Equation 8.5. In order to have good time localization, the wavelets are usually bandpass signals and they decay rapidly towards zero with time. We can also find several other important properties of the wavelet transform and several differences between STFT and the wavelet transform.

The STFT uses a sinusoidal wave as its basis function. These basis functions keep the same frequency over the entire time interval. In contrast, the wavelet transform uses a particular wavelet as its basis function. Hence, wavelets vary in both position and frequency over the time interval. Examples of two basis functions for the sinusoidal wave and wavelet are shown in Figure 8.1(a) and (b), respectively.

The STFT uses a single analysis window. In contrast, the wavelet transform uses a short time window at high frequencies and a long time window at low frequencies. This is referred to as constant Q-factor filtering or relative constant bandwidth frequency analysis. A comparison of the

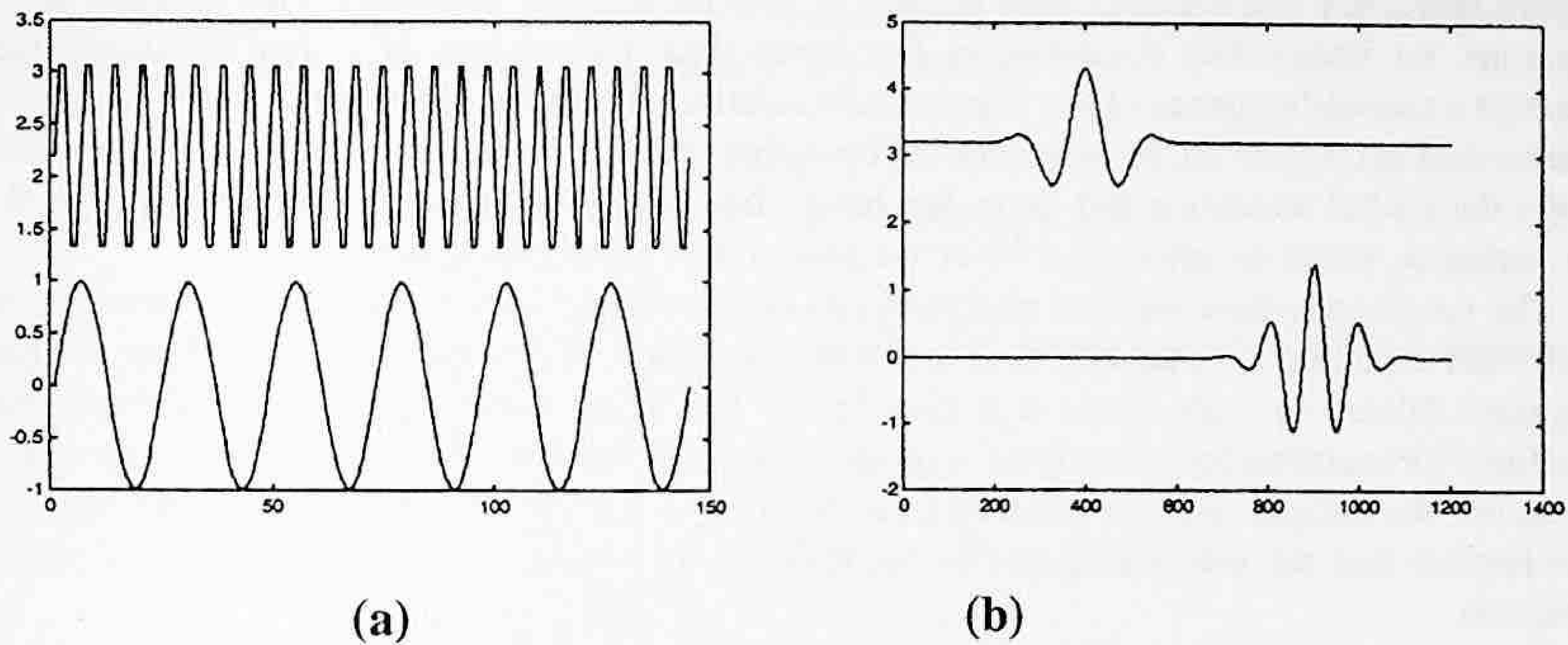


FIGURE 8.1 (a) Two sinusoidal waves, and (b) two wavelets.

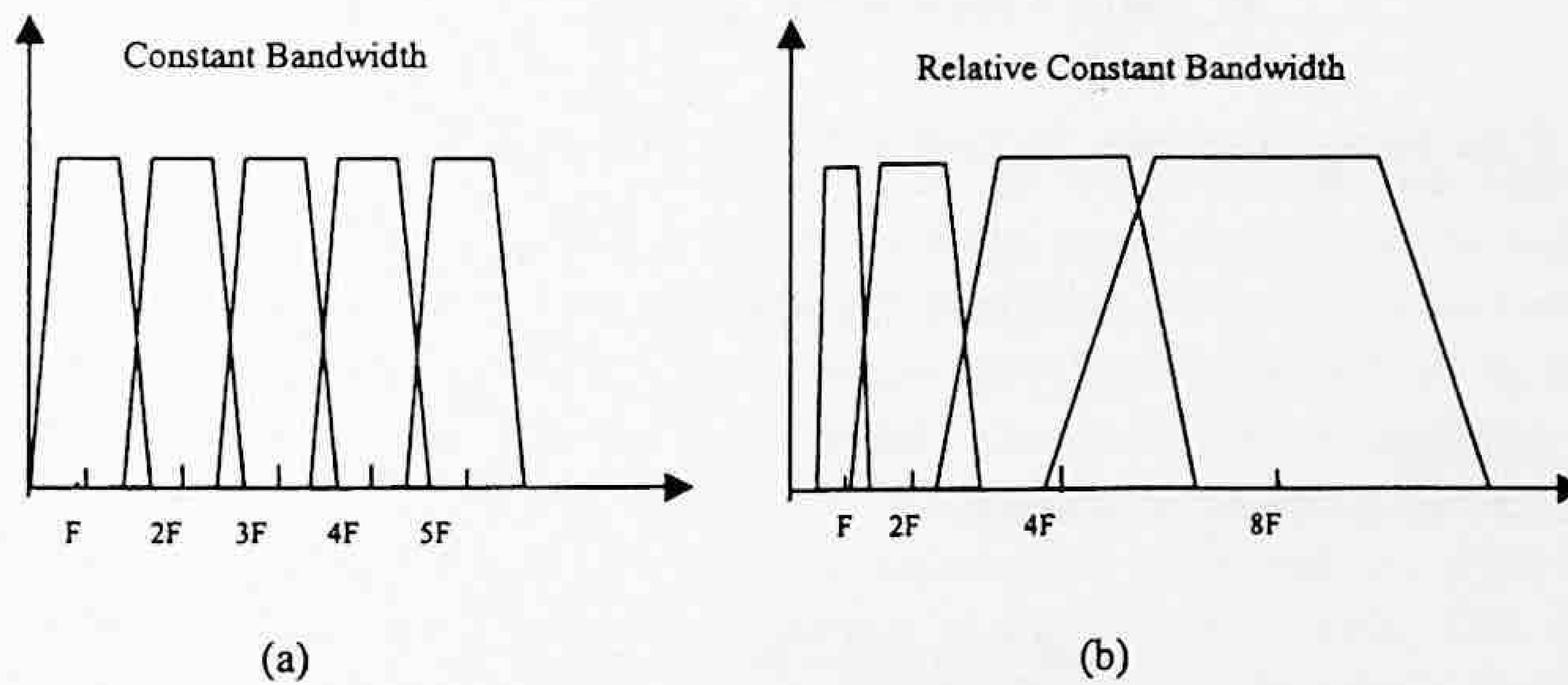


FIGURE 8.2 (a) Constant bandwidth analysis (for Fourier transform), and (b) relative constant bandwidth analysis (for wavelet transform).

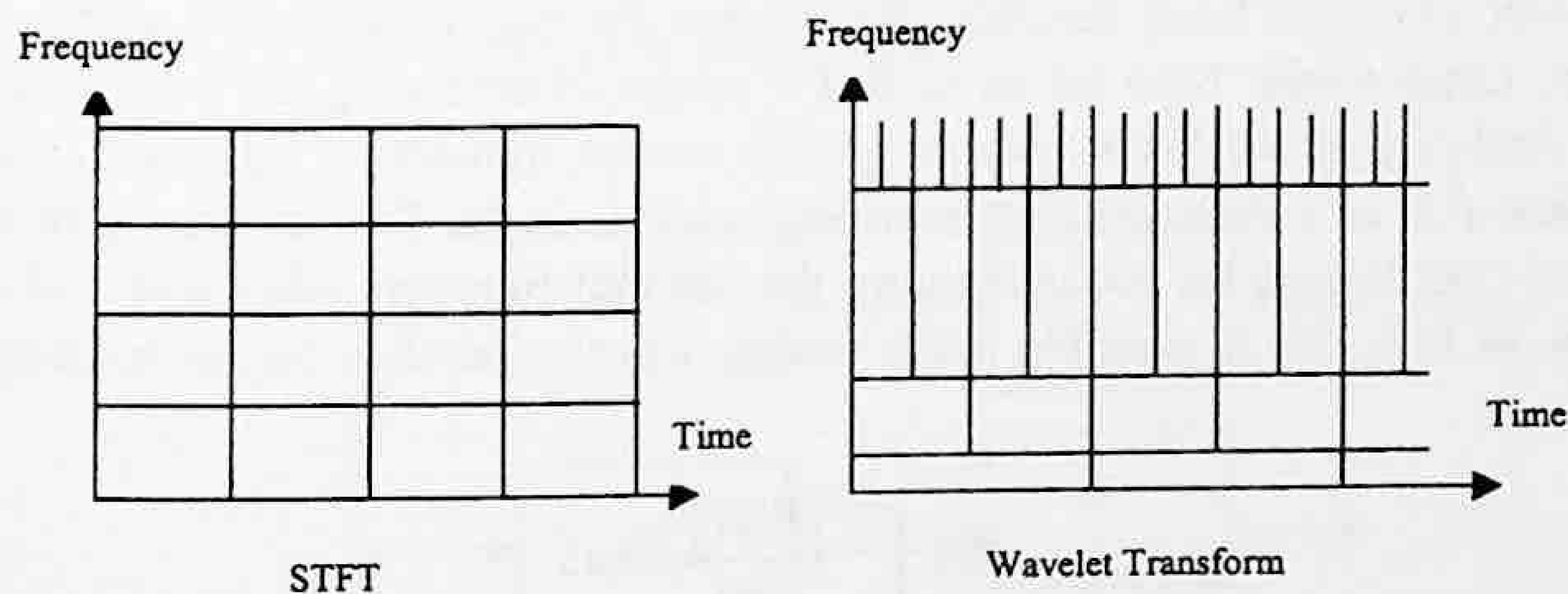


FIGURE 8.3 Comparison of the STFT and the wavelet transform in the time-frequency plane.

constant bandwidth analysis of the STFT and the relative constant bandwidth wavelet transform is shown in Figure 8.2(a) and (b), respectively.

This feature can be further explained with the concept of a time-frequency plane, which is shown in Figure 8.3.

As shown in Figure 8.3, the window size of the STFT in the time domain is always chosen to be constant. The corresponding frequency bandwidth is also constant. In the wavelet transform, the window size in the time domain varies with the frequency. A longer time window is used for

IPR2021-00827
Unified EX1008 Page 197

a lower frequency and a shorter time window is used for a higher frequency. This property is very important for image data compression. For image data, the concept of a time-frequency plane becomes a spatial-frequency plane. The spatial resolution of a digital image is measured with pixels, as described in Chapter 15. To overcome the limitations of DCT-based coding, the wavelet transform allows the spatial resolution and frequency bandwidth to vary in the spatial-frequency plane. With this variation, better bit allocation for active and smooth areas can be achieved.

The continuous-time wavelet transform can be considered as a correlation. For fixed a, it is clear from Equation 8.3 that $W(a,\tau)$ is the cross-correlation of functions $f(t)$ with related wavelet conjugate dilated to scale factor a at time lag τ. This is an important property of the wavelet transform for multiresolution analysis of image data. Since the convolution can be seen as a filtering operation, the integral wavelet transform can be seen as a bank of linear filters acting upon $f(t)$. This implies that the image data can be decomposed by a bank of filters defined by the wavelet transform.

The continuous-time wavelet transform can be seen as an operator. First, it has the property of linearity. If we rewrite $W(a,\tau)$ as $W_{a\tau}[f(t)]$, then we have

$$W_{a\tau}\left[\alpha f(t)+\beta g(t)\right]=\alpha W_{a\tau}\left[f(t)\right]+\beta\ W_{a\tau}\left[g(t)\right] \tag{8.10}$$

where α and β are constant scalars. Second, it has the property of translation:

$$W_{a\tau}\left[f(t-\lambda)\right]=W(a,\tau-\lambda) \tag{8.11}$$

where λ is a time lag.

Finally, it has the property of scaling

$$W_{a\tau}\left[f(t/\alpha)\right]=W(a/\alpha,\tau/\alpha) \tag{8.12}$$

8.1.2 Discrete Wavelet Transform

In the continuous-time wavelet transform, the function $f(t)$ is transformed to a function $W(a,\tau)$ using the wavelet $\psi(t)$ as a basis function. Recall that the two variables a and τ are the dilation and translation, respectively. Now let us to find a means of obtaining the inverse transform, i.e., given $W(a,b)$, find $f(t)$. If we know how to get the inverse transform, we can then represent any arbitrary function $f(t)$ as a summation of wavelets, such as in the Fourier transform and DCT that provide a set of coefficients for reconstructing the original function using sine and cosine as the basis functions. In fact, this is possible if the mother wavelet satisfies the admissibility condition:

$$C=\int_{-\infty}^{+\infty}\frac{\left|\Psi(\omega)\right|^2}{|\omega|}d\omega \tag{8.13}$$

where C is a finite constant and $\Psi(\omega)$ is the Fourier transform of the mother wavelet function $\psi(t)$. Then, the inverse wavelet transform is

$$f(t)=\frac{1}{C}\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty}\frac{1}{|a|^2}W(a,\tau)\psi_{a\tau}(t)da d\tau \tag{8.14}$$

The above results can be extended for two-dimensional signals. If $f(x,y)$ is a two-dimensional function, its continuous-time wavelet transform is defined as:

$$W(a,\tau_x,\tau_y)=\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty}f(x,y)\psi^{*}_{a\tau_x\tau_y}(x,y)dxdy \tag{8.15}$$

where τ_x and τ_y specify the transform in two dimensions. The inverse two-dimensional continuous-time wavelet transform is then defined as:

$$f(x,y)=\frac{1}{C}\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty}\frac{1}{|a|^3}W(a,\tau_x,\tau_y)\psi_{a\tau_x\tau_y}(x,y)dad\tau_x d\tau_y \tag{8.16}$$

where the C is defined as in Equation 8.13 and $\psi(x,y)$ is a two-dimensional wavelet

$$\psi_{a\tau_x\tau_y}(x,y)=\frac{1}{|a|}\psi\left(\frac{x-\tau_x}{a},\frac{y-\tau_y}{a}\right) \tag{8.17}$$

For image coding, the wavelet is used to decompose the image data into wavelets. As indicated in the third property of the wavelet transform, the wavelet transform can be viewed as the cross-correlation of the function $f(t)$ and the wavelets $\psi_{a\tau}(t)$. Therefore, the wavelet transform is equivalent to finding the output of a bank of bandpass filters specified by the wavelets of $\psi_{a\tau}(t)$ as shown in Figure 8.4. This process decomposes the input signal into several subbands. Since each subband can be further partitioned, the filter bank implementation of the wavelet transform can be used for multiresolution analysis (MRA). Intuitively, when the analysis is viewed as a filter bank, the time resolution must increase with the central frequency of the analysis filters. This can be exactly obtained by the scaling property of the wavelet transform, where the center frequencies of the bandpass filters increase as the bandwidth becomes wider. Again, the bandwidth becomes wider by reducing the dilation parameter a. It should be noted that such a multiresolution analysis is consistent with the constant Q-factor property of the wavelet transform. Furthermore, the resolution limitation of the STFT does not exist in the wavelet transform since the time-frequency resolutions in the wavelet transform vary, as shown in Figure 8.2(b).

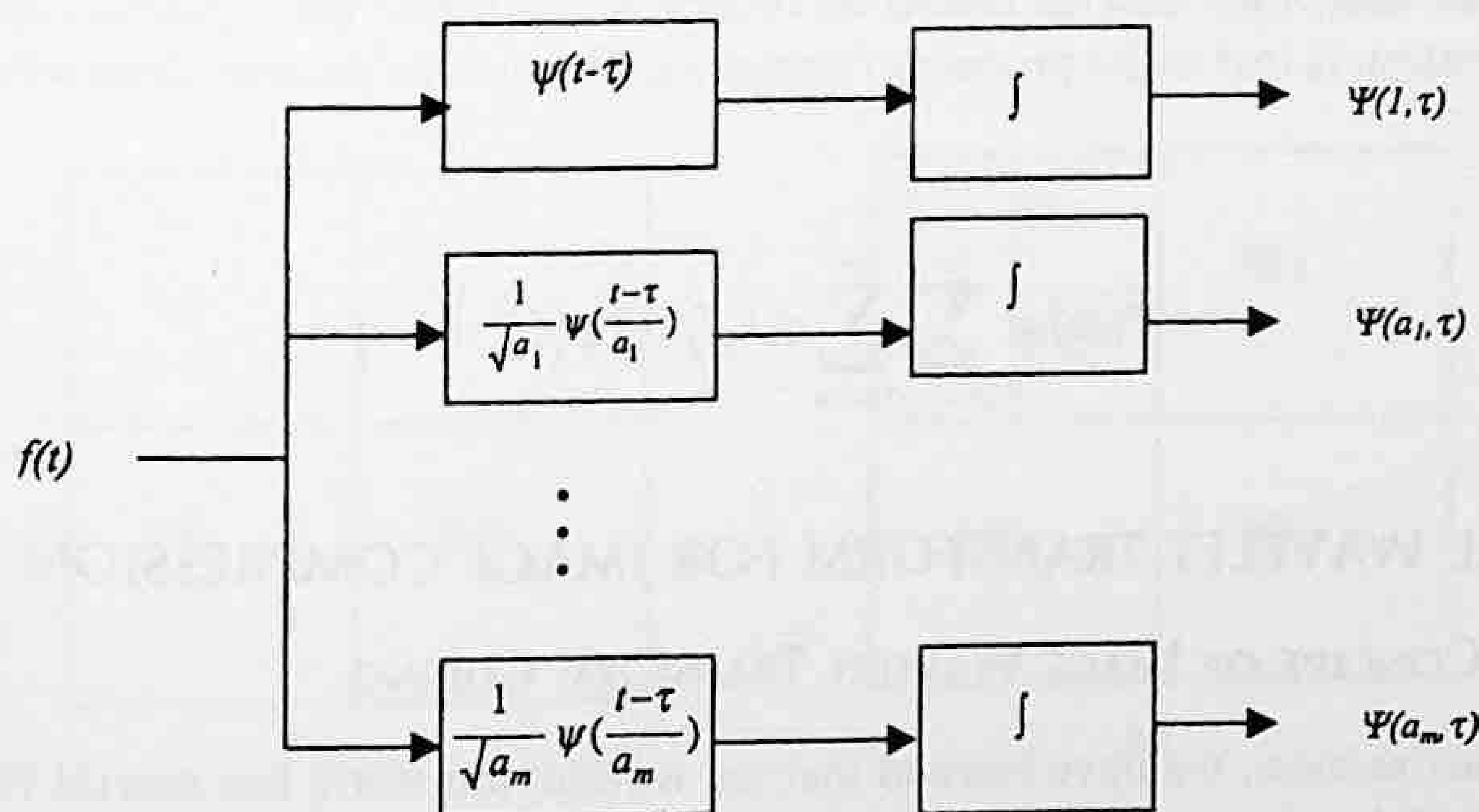


FIGURE 8.4 The wavelet transform implemented with a bank of filters.

For digital image compression, it is preferred to represent $f(t)$ as a discrete superposition sum rather than an integral. With this move to the discrete space, the dilation parameter a in Equation 8.10 takes the values $a = 2^k$ and the translation parameter τ takes the values $\tau = 2^k l$, where both k and l are integers. From Equation 8.4, the discrete version of $\psi_{a\tau}(t)$ becomes:

$$\psi_{kl}(t) = 2^{-\frac{k}{2}}\psi\left(2^{-k}t - l\right) \tag{8.18}$$

Its corresponding wavelet transform can be rewritten as:

$$W(k,l) = \int_{-\infty}^{+\infty} f(t)\psi_{kl}^{*}(t)dt \tag{8.19}$$

and the inverse transform becomes:

$$f(t) = \sum_{k=-\infty}^{+\infty}\sum_{l=-\infty}^{+\infty} d(k,l)2^{-\frac{k}{2}}\psi\left(2^{-k}t - l\right) \tag{8.20}$$

The values of the wavelet transform at those a and τ are represented by $d(k,l)$:

$$d(k,l) = W(k,l)/C \tag{8.21}$$

The $d(k,l)$ coefficients are referred to as the discrete wavelet transform of the function $f(t)$ (Daubechies, 1992; Vetterli and Kovacevic, 1995). It is noted that the discretization so far is only applied to the parameters a and τ; $d(k,l)$ is still a continuous-time function. If the discretization is further applied to the time domain by letting $t = mT$, where m is an integer and T is the sampling interval (without loss of generality, we assume $T = 1$), then the discrete-time wavelet transform is defined as:

$$W_d(k,l) = \sum_{m=-\infty}^{+\infty} f(m)\psi_{kl}^{*}(m) \tag{8.22}$$

Of course, the sampling interval has to be chosen according to the Nyquist sampling theorem so that no information is lost in the process of sampling. The inverse discrete-time wavelet transform is then

$$f(m) = \sum_{m=-\infty}^{+\infty}\sum_{l=-\infty}^{+\infty} d(k,l)2^{-\frac{k}{2}}\psi\left(2^{-k}m - l\right) \tag{8.23}$$

8.2 DIGITAL WAVELET TRANSFORM FOR IMAGE COMPRESSION

8.2.1 BASIC CONCEPT OF IMAGE WAVELET TRANSFORM CODING

From the previous section, we have learned that the wavelet transform has several features that are different from traditional transforms. It is noted from Figure 8.2 that each transform coefficient in the STFT represents a constant interval of time regardless of which band the coefficient belongs to, whereas for the wavelet transform, the coefficients at the course level represent a larger time

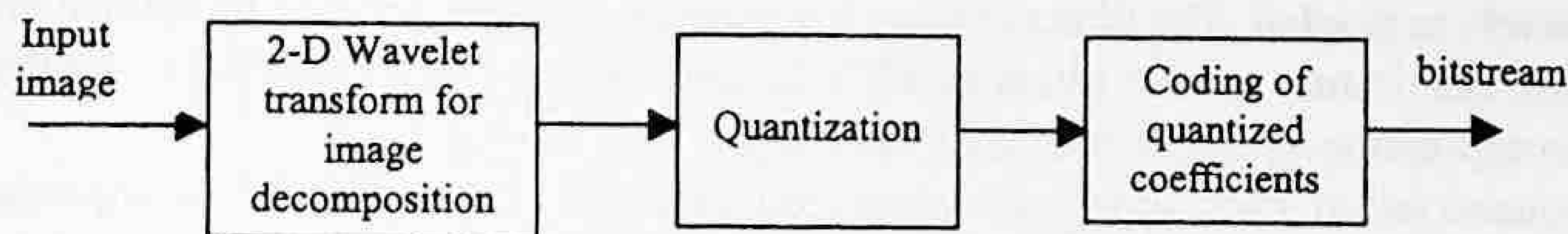


FIGURE 8.5 Block diagram of the image coding with the wavelet transform coding.

interval but a narrower band of frequencies. This feature of the wavelet transform is very important for image coding. In traditional image transform coding, which makes use of the Fourier transform or discrete cosine transform (DCT), one difficult problem is to choose the block size or window width so that statistics computed within that block provide good models of the image signal behavior. The choice of the block size has to be compromised so that it can handle both active and smooth areas. In the active areas, the image data are more localized in the spatial domain, while in the smooth areas the image data are more localized in the frequency domain. With traditional transform coding, it is very hard to reach a good compromise. The main contribution of wavelet transform theory is that it provides an elegant framework in which both statistical behaviors of image data can be analyzed with equal importance. This is because that wavelets can provide a signal representation in which some of the coefficients represent long data lags corresponding to a narrow band or low frequency range, and some of the coefficients represent short data lags corresponding to a wide band or high frequency range. Therefore, it is possible to obtain a good trade-off between spatial and frequency domain with the wavelet representation of image data.

To use the wavelet transform for image coding applications, an encoding process is needed which includes three major steps: image data decomposition, quantization of the transformed coefficients, and coding of the quantized transformed coefficients. A simplified block diagram of this process is shown in Figure 8.5. The image decomposition is usually a lossless process which converts the image data from the spatial domain to frequency domain, where the transformed coefficients are decorrelated. The information loss happens in the quantization step and the compression is achieved in the coding step. To begin the decomposition, the image data are first partitioned into four subbands labeled as LL_1, HL_1, LH_1, and HH_1, as shown in Figure 8.6(a). Each coefficient represents a spatial area corresponding to one-quarter of the original image size. The low frequencies represent a bandwidth corresponding to $0 < |\omega| < \pi/2$, while the high frequencies represent the band $\pi/2 < |\omega| < \pi$. To obtain the next level of decomposition, the LL_1 subband is further decomposed into the next level of four subbands, as shown in Figure 8.6(b). The low frequencies of the second level decomposition correspond to $0 < |\omega| < \pi/4$, while the high frequencies at the second level correspond to $\pi/4 < |\omega| < \pi/2$. This decomposition can be continued

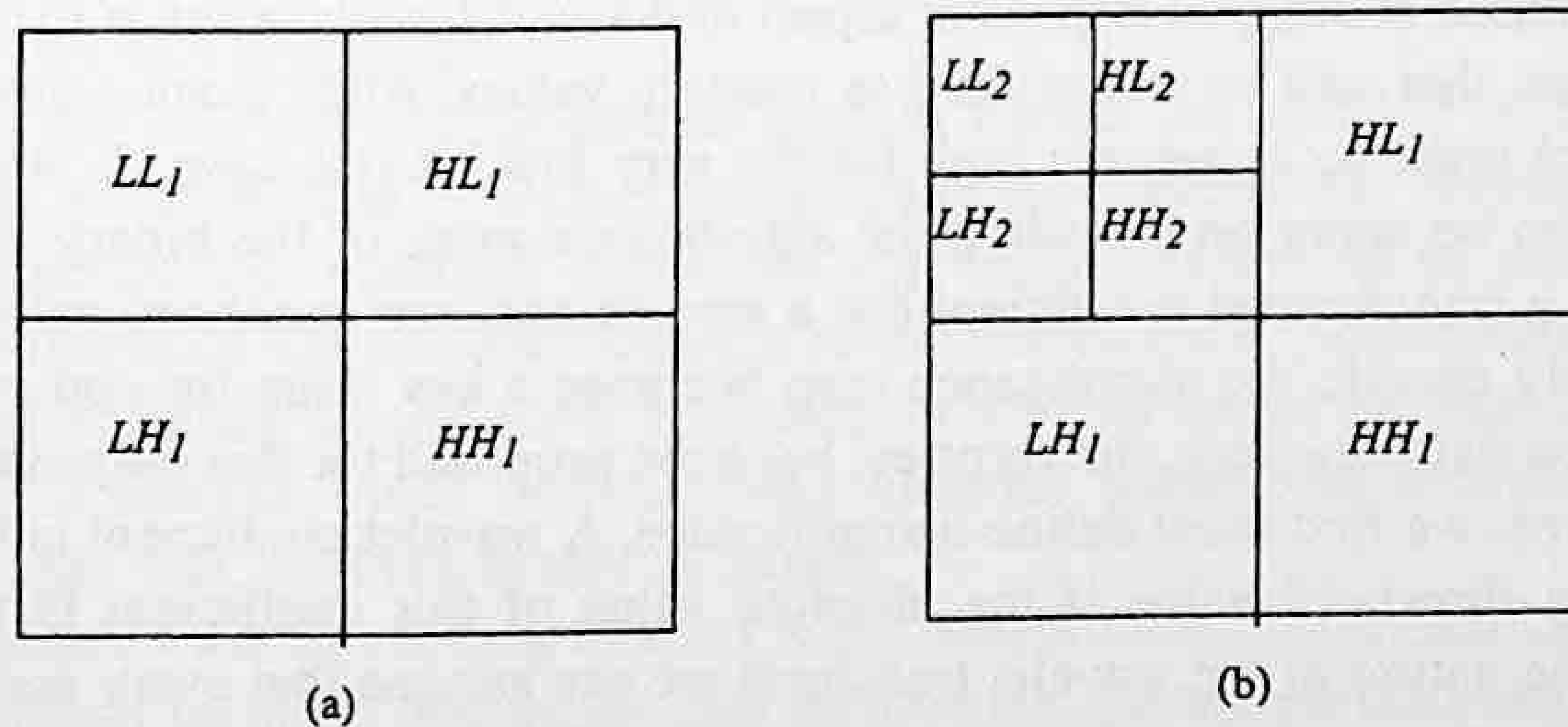


FIGURE 8.6 Two-dimensional wavelet transform. (a) First-level decomposition, and (b) second-level decomposition. (L denotes a low band, H denotes a high band, and the subscript denotes the number of the level. For example, LL_1 denotes the low-low band at level 1.)

to as many levels as needed. The filters used to compute the discrete wavelet transform are generally the symmetric quadrature mirror filters (QMF), as described by Woods (1991). A QMF-pyramid subband decomposition is illustrated in Figure 8.6(b).

During quantization, each subband is quantized differently depending on its importance, which is usually based on its energy or variance (Jayant and Noll, 1984). To reach the predetermined bit rate or compression ratio, coarse quantizers or large quantization steps would be used to quantize the low-energy subbands while the finer quantizers or small quantization steps would be used to quantize the large-energy subbands. This results in fewer bits allocated to those low-energy subbands and more bits for large-energy subbands.

8.2.2 Embedded Image Wavelet Transform Coding Algorithms

As with other transform coding schemes, most wavelet coefficients in the high-frequency bands have very low energy. After quantization, many of these high-frequency wavelet coefficients are quantized to zero. Based on the statistical property of the quantized wavelet coefficients, Huffman coding tables can be designed. Generally, most of the energy in an image is contained in the low-frequency bands. The data structure of the wavelet-transformed coefficients is suitable to exploit this statistical property.

Consider a multilevel decomposition of an image with the discrete wavelet transform, where the lowest levels of decomposition would correspond to the highest-frequency subbands and the finest spatial resolution, and the highest level of decomposition would correspond to the lowest-frequency subband and the coarsest spatial resolution. Arranging the subbands from lowest to highest frequency, we expect a decrease in energy. Also, we expect that if the wavelet-transformed coefficients at a particular level have lower energy, then coefficients at the lower levels or high-frequency subbands, which correspond to the same spatial location, would have smaller energy.

Another feature of the wavelet coefficient data structure is spatial self-similarity across subbands. Several algorithms that have been developed to exploit this and the above-mentioned properties for image coding. Among them, one of the first was proposed by Shapiro (1993) and used an embedded zerotree technique referred to as EZW. Another algorithm is the so-called set partitioning in hierarchical trees (SPIHT) developed by Said and Pearlman (1996). This algorithm also produces an embedded bitstream. The advantage of the embedded coding schemes allows an encoding process to terminate at any point so that a target bit rate or distortion metric can be met exactly. Intuitively, for a given bit rate or distortion requirement a nonembedded code should be more efficient than an embedded code since it has no constraints imposed by embedding requirements. However, embedded wavelet transform coding algorithms are currently the best. The additional constraints do not seem to have deleterious effect. In the following, we introduce the two embedded coding algorithms: the zerotree coding and the set partitioning in hierarchical tree coding.

As with DCT-based coding, an important aspect of wavelet-based coding is to code the positions of those coefficients that will be transmitted as nonzero values. After quantization the probability of the zero symbol must be extremely high for the very low bit rate case. A large portion of the bit budget will then be spent on encoding the significance map, or the binary decision map that indicates whether a transformed coefficient has a zero or nonzero quantized value. Therefore, the ability to efficiently encode the significance map becomes a key issue for coding images at very low bit rates. A new data structure, the zerotree, has been proposed for this purpose (Shapiro, 1993). To describe zerotree, we first must define insignificance. A wavelet coefficient is insignificant with respect to a given threshold value if the absolute value of this coefficient is smaller than this threshold. From the nature of the wavelet transform we can assume that every wavelet transformed at a given scale can be strongly related to a set of coefficients at the next finer scale of similar orientation. More specially, we can further assume that if a wavelet coefficient at a coarse scale is insignificant with respect to the preset threshold, then all wavelet coefficients at finer scales are likely to be insignificant with respect to this threshold. Therefore, we can build a tree with these

IPR2021-00827
Unified EX1008 Page 202

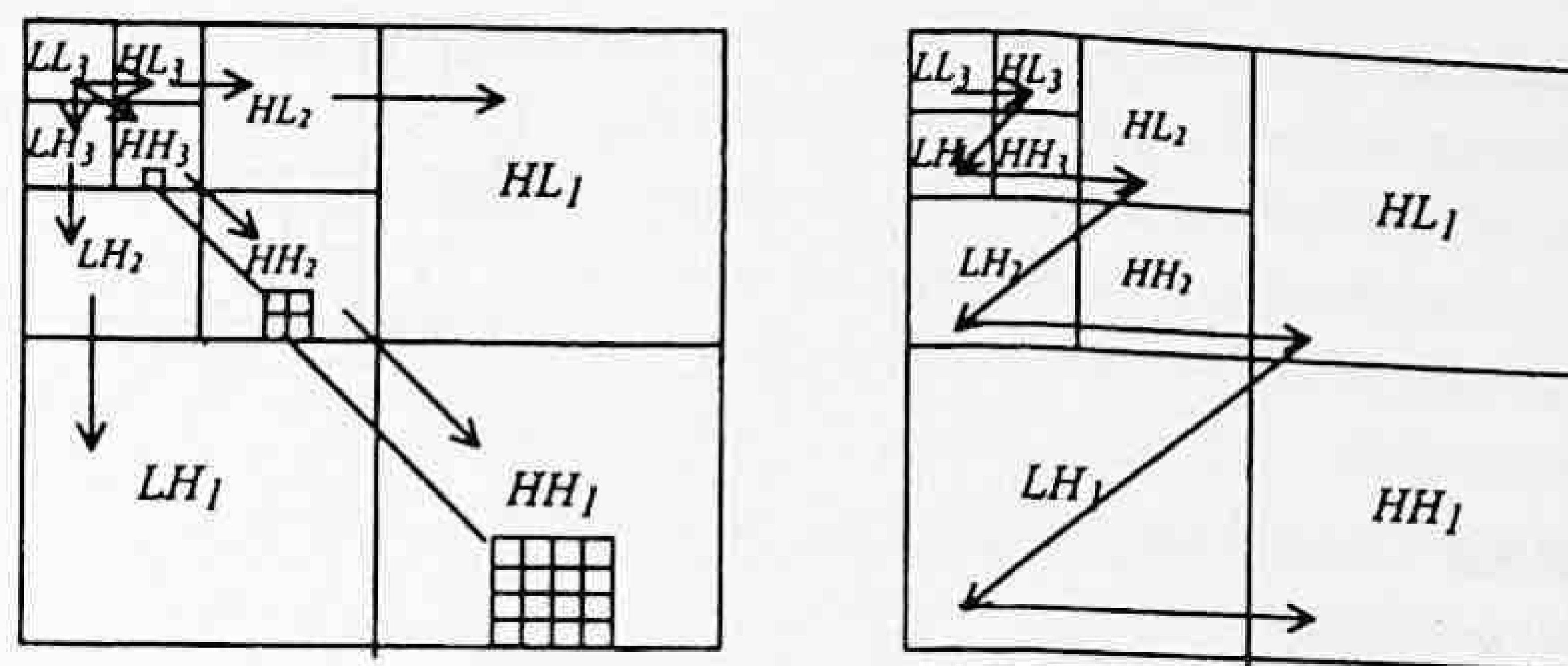


FIGURE 8.7 (Left) Parent-children dependencies of subbands; the arrow points from the subband of the parents to the subband of the children. At top left is the lowest-frequency band. (Right) The scanning order of the subbands for encoding a significance map.

parent-child relationships, such that coefficients at a coarse scale are called parents, and all coefficients corresponding to the same spatial location at the next finer scale of similar orientation are called children. Furthermore, for a parent, the set of all coefficients at all finer scales of similar orientation corresponding to the same spatial location are called descendants. For a QMF-pyramid decomposition the parent-children dependencies are shown in Figure 8.7(a). For a multiscale wavelet transform, the scan of the coefficients begins at the lowest frequency subband and then takes the order of *LL*, *HL*, *LH*, and *HH* from the lower scale to the next higher scale, as shown in Figure 8.7(b).

The zerotree is defined such that if a coefficient itself and all of its descendants are insignificant with respect to a threshold, then this coefficient is considered an element of a zerotree. An element of a zerotree is considered as a zerotree root if this element is not the descendant of a previous zerotree root with respect to the same threshold value. The significance map can then be efficiently represented by a string with three symbols: zerotree root, isolated zero, and significant. The isolated zero means that the coefficient is insignificant, but it has some significant descendant. At the finest scale, only two symbols are needed since all coefficients have no children, thus the symbol for zerotree root is not used. The symbol string is then entropy encoded. Zerotree coding efficiently reduces the cost for encoding the significance map by using self-similarity of the coefficients at different scales. Additionally, it is different from the traditional run-length coding that is used in DCT-based coding schemes. Each symbol in a zerotree is a single terminating symbol, which can be applied to all depths of the zerotree, similar to the end-of-block (EOB) symbol in the JPEG and MPEG video coding standards. The difference between the zerotree and EOB is that the zerotree represents the insignificance information at a given orientation across different scale layers. Therefore, the zerotree can efficiently exploit the self-similarity of the coefficients at the different scales corresponding to the same spatial location. The EOB only represents the insignificance information over the spatial area at the same scale.

In summary, the zerotree-coding scheme tries to reduce the number of bits to encode the significance map, which is used to encode the insignificant coefficients. Therefore, more bits can be allocated to encode the important significant coefficients. It should be emphasized that this zerotree coding scheme of wavelet coefficients is an embedded coder, which means that an encoder can terminate the encoding at any point according to a given target bit rate or target distortion metric. Similarly, a decoder which receives this embedded stream can terminate at any point to reconstruct an image that has been scaled in quality.

Another embedded wavelet coding method is the SPIHT-based algorithm (Said and Pearlman, 1996). This algorithm includes two major core techniques: the set partitioning sorting algorithm and the spatial orientation tree. The set partitioning sorting algorithm is the algorithm that hierarchically divides coefficients into significant and insignificant, from the most significant bit to the least significant bit, by decreasing the threshold value at each hierarchical step for constructing a significance map. At each threshold value, the coding process consists of two passes: the sorting

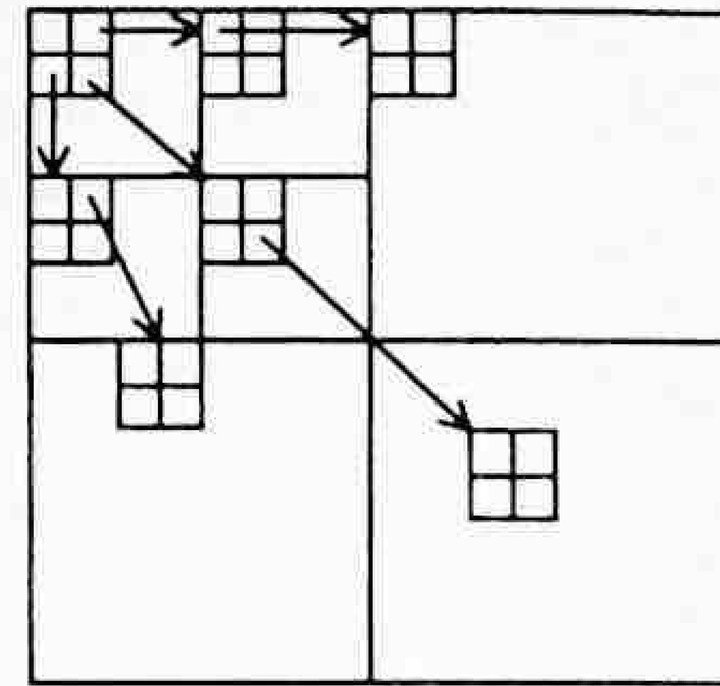


FIGURE 8.8 Relationship between pixels in the spatial orientation tree.

pass and the refinement pass — except for the first threshold that has only the sorting pass. Let $c(i,j)$ represent the wavelet-transformed coefficients and m is an integer. The sorting pass involves selecting the coefficients such that $2^m \le |c(i,j)| \le 2^{m+1}$, with m being decreased at each pass. This process divides the coefficients into subsets and then tests each of these subsets for significant coefficients. The significance map constructed in the procedure is tree-encoded. The significant information is store in three ordered lists: list of insignificant pixels (LIP), list of significant pixels (LSP), and list of insignificant sets (LIS). At the end of each sorting pass, the LSP contains the coordinates of all significant coefficients with respect to the threshold at that step. The entries in the LIS can be one of two types: type A represents all its descendants, type B represents all its descendants from its grandchildren onward. The refinement pass involves transmitting the mth-most significant bit of all the coefficients with respect to the threshold, 2^{m+1}.

The idea of a spatial orientation tree is based on the following observation. Normally, among the transformed coefficients most of the energy is concentrated in the low frequencies. For the wavelet transform, when we move from the highest to the lowest levels of the subband pyramid the energy usually decreases. It is also observed that there exists strong spatial self-similarity between subbands in the same spatial location such as in the zerotree case. Therefore, a spatial orientation tree structure has been proposed for the SPIHT algorithm. The spatial orientation tree naturally defines the spatial relationship on the hierarchical pyramid as shown in Figure 8.8.

During the coding, the wavelet-transformed coefficients are first organized into spatial orientation trees as in Figure 8.8. In the spatial orientation tree, each pixel (i,j) from the former set of subbands is seen as a root for the pixels $(2i, 2j)$, $(2i+1, 2j)$, $(2i,2j+1)$, and $(2i+1, 2j+1)$ in the subbands of the current level. For a given n-level decomposition, this structure is used to link pixels of the adjacent subbands from level n until to level 1. In the highest-level n, the pixels in the low-pass subband are linked to the pixels in the three high-pass subbands at the same level. In the subsequent levels, all the pixels of a subband are involved in the tree-forming process. Each pixel is linked to the pixels of the adjacent subband at the next lower level. The tree stops at the lowest level.

The implementation of the SPIHT algorithm consists of four steps: initialization, sorting pass, refinement pass, and quantization scale update. In the initialization step, we find an integer $m = \lfloor \log_2(\max_{(i,j)}\{|c(i,j)|\}) \rfloor$. Here $\lfloor \ \rfloor$ represent an operation of obtaining the largest integer less than $|c(i,j)|$. The value of m is used for testing the significance of coefficients and constructing the significance map. The LIP is set as an empty list. The LIS is initialized to contain all the coefficients in the low-pass subbands that have descendants. These coefficients can be used as roots of spatial trees. All these coefficients are assigned to be of type A. The LIP is initialized to contain all the coefficients in the low-pass subbands.

In the sorting pass, each entry of the LIP is tested for significance with respect to the threshold value 2^m. The significance map is transmitted in the following way. If it is significant, a "1" is transmitted, a sign bit of the coefficient is transmitted, and the coefficient coordinates are moved to the LSP. Otherwise, a "0" is transmitted. Then, each entry of the LIS is tested for finding the significant descendants. If there are none, a "0" is transmitted. If the entry has at least one significant descendant, then a "1" is transmitted and each of the immediate descendants are tested for significance. The significance map for the immediate descendants is transmitted in such a way that if it

is significant, a "1" plus a sign bit are transmitted and the coefficient coordinates are appended to the LSP. If it is not significant, a "0" is transmitted and the coefficient coordinates are appended to the LIP. If the coefficient has more descendants, then it is moved to the end of the LIS as an entry of type B. If an entry in the LIS is of type B, then its descendants are tested for significance. If at least one of them is significant, then this entry is removed from the list, and its immediate descendants are appended to the end of the list of type A. For the refinement pass, the mth-most significant bit of the magnitude of each entry of the LSP is transmitted except those in the current sorting pass. For the quantization scale update step, m is decreased by 1 and the procedure is repeated from the sorting pass.

8.3 WAVELET TRANSFORM FOR JPEG-2000

8.3.1 Introduction to JPEG-2000

Most image coding standards so far have exploited the DCT as their core technique for image decomposition. However, recently there has been a noticeable change. The wavelet transform has been adopted by MPEG-4 for still image coding (mpeg4). Also, JPEG-2000 is considering using the wavelet transform as its core technique for the next generation of the still image coding standard (jpeg2000 vm). This is because the wavelet transform can provide not only excellent coding efficiency, but also good spatial and quality scalable functionality. JPEG-2000 is a new type of image compression system under development by Joint Photographic Experts Group for still image coding. This standard is intended to meet a need for image compression with great flexibility and efficient interchangeability. JPEG-2000 is also intended to offer unprecedented access into the image while still in compressed domain. Thus, images can be accessed, manipulated, edited, transmitted, and stored in a compressed form. As a new coding standard, the detailed requirements of JPEG-2000 include:

Low bit-rate compression performance: JPEG-2000 is required to offer excellent coding performance at bit rates lower than 0.25 bits per pixel for highly detailed gray-bits per level images since the current JPEG (10918-1) cannot provide satisfactory results at this range of bit rates. This is the primary feature of JPEG-2000.

Lossless and lossy compression: it is desired to provide lossless compression naturally in the course of progressive decoding. This feature is especially important for medical image coding where the loss is not always allowed. Also, other applications such as high-quality image archival systems and network applications desire to have the functionality of lossless reconstruction.

Large images: currently, the JPEG image compression algorithm does not allow for images greater than 64K by 64K without tiling.

Single decomposition architecture: the current JPEG standard has 44 modes; many of these modes are for specific applications and not used by the majority of JPEG decoders. It is desired to have a single decomposition architecture that can encompass the interchange between applications.

Transmission in noisy environments: it is desirable to consider error robustness while designing the coding algorithm. This is important for the application of wireless communication. The current JPEG has provision for restart intervals, but image quality suffers dramatically when bit errors are encountered.

Computer-generated imagery: the current JPEG is optimized for natural imagery and does not perform well on computer-generated imagery or computer graphics.

Compound documents: the new coding standard is desired to be capable of compressing both continuous-tone and bilevel images. The coding scheme can compress and decompress images from 1 bit to 16 bits for each color component. The current JPEG standard does not work well for bilevel images.

IPR2021-00827
Unified EX1008 Page 205

Progressive transmission by pixel accuracy and resolution: progressive transmission that allows images to be transmitted with increasing pixel accuracy or spatial resolution is important for many applications. The image can be reconstructed with different resolutions and pixel accuracy as needed for different target devices such as in World Wide Web applications and image archiving.

Real-time encoding and decoding: for real-time applications, the coding scheme should be capable of compressing and decompressing with a single sequential pass. Of course, optimal performance cannot be guaranteed in this case.

Fixed rate, fixed size, and limited workspace memory: the requirement of fixed bit rate allows the decoder to run in real time through channels with limited bandwidth. The limited memory space is required by the hardware implementation of decoding.

There are also some other requirements such as backwards compatibility with JPEG, open architecture for optimizing the system for different image types and applications, interface with MPEG-4, and so on. All these requirements are seriously being considered during the development of JPEG-2000. However, it is still too early to comment whether all targets can be reached at this moment. There is no doubt, though, that the basic requirement on the coding performance at very low bit rate for still image coding will be achieved by using wavelet-based coding as the core technique instead of DCT-based coding.

8.3.2 Verification Model of JPEG-2000

Since JPEG-2000 is still awaiting finalization, we introduce the techniques that are very likely to be adopted by the new standard. As in other standards such as MPEG-2 and MPEG-4, the verification model (VM) plays an important role during the development of standards. This is because the VM or TM (test model for MPEG-2) is a platform for verifying and testing the new techniques before they are adopted as standards. The VM is updated by completing a set of core experiments from one meeting to another. Experience has shown that the decoding part of the final version of VM is very close to the final standard. Therefore, in order to give an overview of the related wavelet transform parts of the JPEG-2000, we start to introduce the newest version of JPEG-2000 VM (jpeg2000 vm). The VM of JPEG-2000 describes the encoding process, decoding process, and the bitstream syntax, which eventually completely defines the functionality of the existing JPEG-2000 compression system.

The newest version of the JPEG-2000 verification model, currently VM 4.0, was revised on April 22, 1999. In this VM, the final convergence has not been reached, but several candidates have been introduced. These techniques include a DCT-based coding mode, which is currently the baseline JPEG, and a wavelet-based coding mode. In the wavelet-based coding mode, several algorithms have been proposed: overlapped spatial segmented wavelet transform (SSWT), non-overlapped SSWT, and the embedded block-based coding with optimized truncation (EBCOT). Among these techniques, and according to current consensus, EBCOT is a very likely candidate for adoption into the final JPEG-2000 standard.

The basic idea of EBCOT is the combination of block coding with wavelet transform. First, the image is decomposed into subbands using the wavelet transform. The wavelet transform is not restricted to any particular decomposition. However, the Mallat wavelet provides the best compression performance, on average, for natural images; therefore, the current bitstream syntax is restricted to the standard Mallat wavelet transform in VM 4.0. After decomposition, each subband is divided into 64×64 blocks, except at image boundaries where some blocks may have smaller sizes. Every block is then coded independently. For each block, a separate bitstream is generated without utilizing any information from other blocks. The key techniques used for coding include an embedded quad-tree algorithm and fractional bit-plane coding.

B_i^1	B_i^2	B_i^3	B_i^4
B_i^5	B_i^6	B_i^7	B_i^8
B_i^9	B_i^{10}	B_i^{11}	B_i^{12}
B_i^{13}	B_i^{14}	B_i^{15}	B_i^{16}

FIGURE 8.9 Example of sub-block partitioning for a block of 64 × 64.

The idea of an embedded quad-tree algorithm is that it uses a single bit to represent whether or not each leading bit-plane contains any significant samples. The quad-tree is formed in the following way. The subband is partitioned into a basic block. The basic block size is 64 × 64. Each basic block is further partitioned into 16 × 16 sub-blocks, as shown in Figure 8.9. Let $\sigma^j(B_i^k)$ denote the significance of sub-block, B_i^k(k is the kth sub-block as shown in Figure 8.9), in jth bit plane of ith block. If one or more samples in the sub-block have the magnitude greater than 2^j, then $\sigma^j(B_i^k) = 1$; otherwise, $\sigma^j(B_i^k) = 0$. For each bit-plane, the information concerning the significant sub-blocks is first encoded. All other sub-blocks can then be bypassed in the remaining coding procedure for that bit-plane. To specify the exact coding sequence, we define a two-level quad-tree for the block size of 64 × 64 and sub-block size of 16 × 16. The level-1 quads, $Q_i^l[k]$, consist of four sub-blocks, B_i^1, B_i^2, B_i^3, B_i^4 from Figure 8.9. In the same way, we define level-2 quads, $Q_i^2[k]$, to be 2 × 2 groupings of level-1 quads. Let $\sigma^j(Q_i^l[k])$ denote the significance of the level-1 quad, $Q_i^l[k]$, in jth bit-plane. If at least one member sub-block is significant in the jth bit-plane, then $\sigma^j(Q_i^l[k]) = 1$; otherwise, $\sigma^j(Q_i^l[k]) = 0$. At each bit-plane, the quad-tree coder visits the level-2 quad first, followed by level-1 quads. When visiting a particular quad, $Q_i^L[k]$(L = 1 or 2, it is the number of the level), the coder sends the significance of each of the four child quads, $\sigma^j(Q_i^L[k])$, or sub-blocks, $\sigma^j(B_i^k)$, as appropriate, except if the significance value can be deduced from the decoder. Under the following three cases, the significance may be deduced by the decoder: (1) the relevant quad or sub-block was significant in the previous bit-plane; (2) the entire sub-block is insignificant; or (3) this is the last child or sub-block visited in $Q_i^L[k]$ and all previous quads or sub-blocks are insignificant.

The idea of bit-plane coding is to entropy code the most significant bit first for all samples in the sub-blocks and to send the resulting bits. Then, the next most significant bit will be coded and sent, this process will be continued until all bit-planes have been coded and sent. This kind of bitstream structure can be used for robust transmission. If the bitstream is truncated due to a transmission error or some other reason, then some or all the samples in the block may lose one or more least significant bits. This will be equivalent to having used a coarser quantizer for the relevant samples and we can still obtain a reduced-quality reconstructed image. The idea of fractional bit-plane coding is to code each bit-plane with four passes: a forward significance propagation pass, a backward significance propagation pass, a magnitude refinement pass, and a normalization pass. For the technical details of fractional bit-plane coding, the interested readers can refer to the VM of JPEG-2000 (jpeg2000 vm).

Finally, we briefly describe the optimization issue of EBCOT. The encoding optimization algorithm is not a part of the standard, since the decoder does not need to know how the encoder generates the bitstream. From the viewpoint of the standard, the only requirement from the decoder to the encoder is that the bitstream must be compliant with the syntax of the standard. However, from the other side, the bitstream syntax could always be defined to favor certain coding algorithms for generating optimized bitstreams. The optimization algorithm described here is justified only if the distortion measure adopted for the code blocks is additive. That is, the final distortion, D, of the whole reconstructed image should satisfy

$$D = \sum D_i^{Ti} \tag{8.24}$$

where D_i is the distortion for block B_i and T_i is the truncation point for B_i. Let R be the total number of bits for coding all blocks of the image for a set of truncation point T_i, then

$$R = \sum R_i^{Ti} \tag{8.25}$$

where R_i^{Ti} are the bits for coding block B_i. The optimization process wishes to find the suitable set of T_i values, which minimizes D subject to the constraint $R \le R_{max}$. R_{max} is the maximum number of bits assigned for coding the image. The solution is obtained by the method of Lagrange multipliers:

$$L = \sum \left(R_i^{Ti} - \lambda D_i^{Ti} \right) \tag{8.26}$$

where the value λ must be adjusted until the rate obtained by the truncation points, which minimize the value of L, satisfies $R = R_{max}$. From Equation 8.26, we have a separate trivial optimization problem for each individual block. Specially, for each block, B_i, we find the truncation point, T_i, which minimizes the value $(R_i^{Ri} - \lambda D_i^{Ti})$. This can be achieved by finding the slope turning points of rate distortion curves. In the VM, the set of truncation points and the slopes of rate distortion curves are computed immediately after each block is coded, and we only store enough information to later determine the truncation points which correspond to the slope turning points of rate distortion curves. This information is generally much smaller than the bitstream which is stored for the block itself. Also, the search for the optimal λ is extremely fast and occupies a negligible portion of the overall computation time.

8.4 SUMMARY

In this chapter, image coding using the wavelet transform has been introduced. First, an overview of wavelet theory was given, and second, the principles of image coding using wavelet transform have been presented. Additionally, two particular embedded image coding algorithms have been explained, namely, the embedded zerotree and set partitioning in hierarchical trees. Finally, the new standard for still image coding, JPEG-2000, which may adopt the wavelet as its core technique, has been described.

8.5 EXERCISES

8-1. For a given function, the Mexican hat wavelet,

$$f(t) = \begin{cases} 1, & for\ |t| \le 1, \\ 0, & otherwise \end{cases}$$

Use Equations 8.3 and 8.4 to derive a closed-form expression for the continuous wavelet transform, $\psi_{ab}(t)$.

8-2. Consider the dilation equation

$$\varphi(t) = \sqrt{2} \sum_k h(k) \varphi(2t - k)$$

How does $\varphi(t)$ change if $h(k)$ is shifted? Specifically, let $g(k) = h(n-l)$

$$u(t) = \sqrt{2}\sum_k g(k)u(2t-k)$$

How does $u(t)$ relate to $\varphi(t)$?

8-3. Let $\varphi_a(t)$ and $\varphi_b(t)$ be two scaling functions generated by the two scaling filters $h_a(k)$ and $h_b(k)$. Show that the convolution $j_a(t)*\ j_b(t)$ satisfies a dilation equation with $h_a(k)*\ h_b(k)/\sqrt{2}$.

8-4. In the applications of denoising and image enhancement, how can the wavelet transform improve the results?

8-5. For a given function

$$f(t) = \begin{cases} 0 & t < 0 \\ t & 0 \le t < 1 \\ 1 & t \ge 1 \end{cases}$$

show that the wavelet transform of $f(t)$ will be

$$W(a,b) = \mathrm{sgn}\,\frac{2f\left(b+\frac{a}{2}\right) - f(b) - f(b+a)}{\sqrt{|a|}}$$

where sgn(x) is the signum function defined as

$$\mathrm{sgn}(x) = \begin{cases} -1 & t < 0 \\ 1 & t > 0 \\ 0 & t = 0 \end{cases}$$

REFERENCES

Cohen, L. Time-Frequency Distributions — A Review, *Proc. IEEE,* Vol. 77, No. 7, July 1989, pp. 941-981.

Daubechies, I. *Ten Lectures on Wavelets,* CBMS Series, Philadelphia, SIAM, 1992.

Grossman, A. and J. Morlet, Decompositions of hard functions into square integrable wavelets of constant shape, *SIAM J. Math. Anal.,* 15(4), 723-736, 1984.

Jayant, N. S. and P. Noll, *Digital Coding of Waveforms,* Englewood Cliffs, NJ: Prentice-Hall, 1984.

jpeg2000 vm, JPEG-2000 Verification Model 4.0 (Tech. description), sc29wg01 N1282, April 22, 1999.

mpeg4, ISO/IEC 14496-2, Coding of Audio-Visual Objects, Nov. 1998.

Said, A. and W. A. Pearlman, A new fast and efficient image codec based on set partitioning in hierarchical trees, *IEEE Trans. Circuits Syst. Video Technol.,* 243-250, 1996.

Shapiro, J. Embedded image coding using zerotrees of wavelet coefficients, *IEEE Trans. Signal Process.,* 3445-3462, Dec. 1993.

Vetterli, M. and J. Kovacevic, *Wavelets and Subband Coding,* Englewood Cliffs, NJ: Prentice-Hall, 1995.

Woods, J., Ed., *Subband Image Coding,* Kluwer Academic Publishers, 1991.

IPR2021-00827
Unified EX1008 Page 210