Imperial College of Science Technology and Medicine

Department of Electrical and Electronic Engineering

Digital Image Processing

PART 4

IMAGE COMPRESSION

LOSSLESS COMPRESSION

Academic responsible

Dr. Tania STATHAKI

Room 812

Ext. 46229

Email: t.stathaki@ic.ac.uk

METHODS AND STANDARDS FOR LOSSLESS COMPRESSION

1 PRELIMINARIES

Lossless compression refers to compression methods for which the original uncompressed data set can be recovered exactly from the compressed stream. The need for lossless compression arises from the fact that many applications, such as the compression of digitized medical data, require that no loss be introduced from the compression method. Bitonal image transmission via a facsimile device also imposes such requirements. In recent years, several compression standards have been developed for the lossless compression of such images. We discuss these standards later. In general, even when lossy compression is allowed, the overall compression scheme may be a combination of a lossy compression process followed by a lossless compression process. Various image, video, and audio compression standards follow this model, and several of the lossless compression schemes used in these standards are described in this section.

The general model of a lossless compression scheme is as depicted in the following figure.

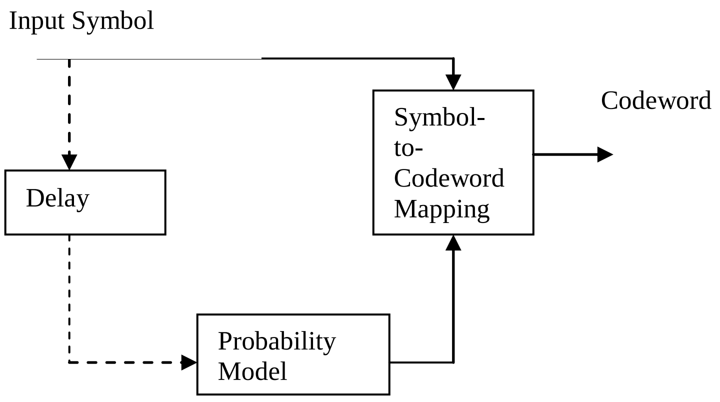


Figure 1.1: A generic model for lossless compression

Given an input set of symbols, a modeler generates an estimate of the probability distribution of the input symbols. This probability model is then used to map symbols into codewords. The combination of the probability modeling and the symbol-to-codeword mapping functions is usually referred to as **entropy coding**. The key idea of entropy coding is to use short codewords for symbols that occur

with high probability and long codewords for symbols that occur with low probability.
The probability model can be derived either from the input data or from a priori assumptions about the data. Note that, for decodability, the same model must also be generated by the decoder. Thus, if the model is dynamically estimated from the input data, causality constraints require a delay function between the input and the modeler. If the model is derived from a priori assumptions, then the delay block is not required; furthermore, the model function need not have access to the input symbols. The probability model does not have to be very accurate, but the more accurate it is, the better the compression will be. Note that, compression is not always guaranteed. If the probability model is wildly inaccurate, then the output size may even expand. However, even then the original input can be recovered without any loss.
Decompression is performed by reversing the flow of operations shown in the above Figure 1.1. This decompression process is usually referred to as **entropy decoding**.

Message-to-Symbol Partitioning

As noted before, entropy coding is performed on a symbol by symbol basis. Appropriate partitioning of the input messages into symbols is very important for efficient coding. For example, typical images have sizes from 256×256 pixels to 64000×64000 pixels. One could view one instance of a 256×256 multi-frame image as a single message, $256^2 = 65536$ long; however, it is very difficult to provide probability models for such long symbols. In practice, we typically view any image as a string of symbols. In the case of a 256×256 image, if we assume that each pixel takes values between zero and 255, then this image can be viewed as a sequence of symbols drawn from the alphabet $0,1,2,\ldots,255$. The modeling problem now reduces to finding a good probability model for the 256 symbols in this alphabet.

For some images, one might partition the data set even further. For instance, if we have an image with 12 bits per pixel, then this image can be viewed as a sequence of symbols drawn from the alphabet $0,1,\ldots,4095$. Hardware and/or software implementations of the lossless compression methods may require that data be processed in $8-, 16-, 32-,$ or $64-$bit units. Thus, one approach might be to take the stream of $12-$bit pixels and artificially view it as a sequence of $8-$bit symbols. In this case, we have reduced the alphabet size. This reduction compromises the achievable compression ratio; however, the data are matched to the processing capabilities of the computing element.

Other data partitions are also possible; for instance, one may view the data as a stream of $24-$bit symbols. This approach may result in higher compression since we are combining two pixels into one symbol. In general, the partitioning of the data into blocks, where a block is composed of several input units, may result in higher compression ratios, but also increases the coding complexity.

Differential Coding

Another preprocessing technique that improves the compression ratio is differential coding. Differential coding skews the symbol statistics so that the resulting distribution is more amenable to compression. Image data tend to have strong inter-pixel correlation. If, say, the pixels in the image are in the order $x_1, x_2, x_3, \ldots, x_N$, then instead of compressing these pixels, one might process the sequence of differentials $y_i = x_i - x_{i-1}$, where $i = 1,2,\ldots,N$, and $x_0 = 0$. In compression terminology, y_i is referred to as the **prediction residual** of x_i. The notion of compressing the prediction residual instead of x_i is used in all the image and video compression standards. For images, a typical probability distribution for x_i and the resulting distribution for y_i are shown in Figure 1.2.

Let symbol s_i have a probability of occurrence p_i. From coding theory, the ideal symbol-to-codeword mapping function will produce a codeword requiring $\log_2(1/p_i)$ bits. A distribution close to uniform for p_i $(p_i \approx 1/255)$, such as the one shown in the left plot of Figure 1.2, will result in codewords that on the average require eight bits; thus, no compression is achieved. On the other hand, for a skewed probability distribution, such as the one shown in the right plot of Figure 1.2, the **symbol-to-codeword mapping function** can on the average yield codewords requiring less than eight bits per symbol and thereby achieve compression.

We will understand these concepts better in the following Huffman encoding section.

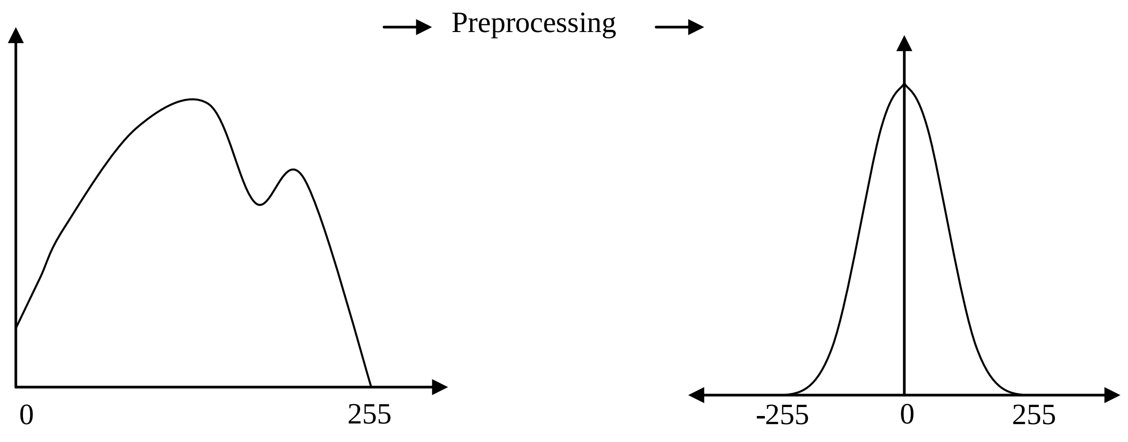


Figure 1.2: Typical distribution of pixel values for x_i and y_i. Here, the pixel values are shown on the horizontal axis and the corresponding probability of occurrence is shown on the

vertical axis.

2 HUFFMAN ENCODING

In 1952, D. A. Huffman developed a code construction method that can be used to perform lossless compression. In Huffman coding, the modeling and the symbol-to-codeword mapping functions of Figure 1.1 are combined into a single process. As discussed earlier, **the input data are partitioned into a sequence of symbols** so as to facilitate the modeling process. In most image and video compression applications, the size of the alphabet composing these symbols is restricted to at most 64000 symbols. The Huffman code construction procedure evolves along the following parts:

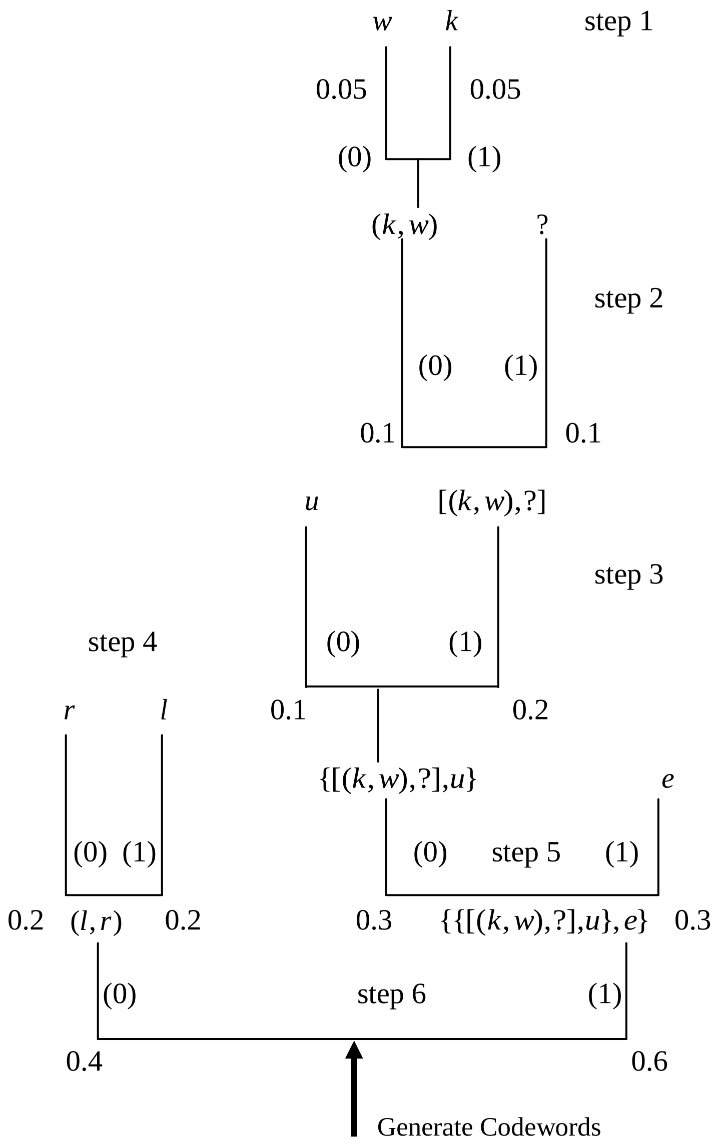
1. Order the symbols according to their probabilities.
   For Huffman code construction, the frequency of occurrence of each symbol must be known a priori. In practice, the frequency of occurrence can be estimated from a training set of data that is representative of the data to be compressed in a lossless manner. If, say, the alphabet is composed of N distinct symbols $s_1, s_2, s_3, \ldots, s_N$ and the probabilities of occurrence are $p_1, p_2, p_3, \ldots, p_N$, then the symbols are rearranged so that $p_1 \geq p_2 \geq p_3 \ldots \geq p_N$.

2. Apply a contraction process to the two symbols with the smallest probabilities.
   Suppose the two symbols are s_{N-1} and s_N. We replace these two symbols by a hypothetical symbol, say, $H_{N-1} = (s_{N-1}, s_N)$ that has a probability of occurrence $p_{N-1} + p_N$. Thus, the new set of symbols has $N-1$ members $s_1, s_2, s_3, \ldots, s_{N-2}, H_{N-1}$.

3. We repeat the previous part 2 until the final set has only one member.

The recursive procedure in part 2 can be viewed as the construction of a binary tree, since at each step we are merging two symbols. At the end of the recursion process all the symbols $s_1, s_2, s_3, \ldots, s_N$ will be leaf nodes of this tree. The codeword for each symbol s_i is obtained by traversing the binary tree from its root to the leaf node corresponding to s_i.

We illustrate the code construction process with the following example depicted in Figure 2.1. The input data to be compressed is composed of symbols in the alphabet $k, l, u, w, e, r, ?$. First we sort the probabilities. In Step 1, we merge the two symbols k and w to form the new symbol (k, w). The probability of occurrence for the new symbol is the sum of the probabilities of occurrence for k and w. We sort the probabilities again and perform the merge on the pair of least frequently occurring symbols which are now the symbols (k, w) and ?. We repeat this process through Step 6. By visualizing this process as a binary tree as shown in this figure and traversing the process from the bottom of the tree to the top, one can determine the codewords for each symbol. For example, to reach the symbol u from the root of the tree, one traverses nodes that were assigned the bits 1,0 and 0. Thus, the codeword for u is 100.



	Step 1	**Step 2**	**Step 3**	**Step 4**	**Step 5**	**Step 6**
k 0.05	e 0.3	e 0.3	e 0.3	e 0.3	(l, r) 0.4	$\{\{[(k, w), ?], u\}, e\}$ 0.6
l 0.2	l 0.2	l 0.2	l 0.2	$\{[(k, w), ?], u\}$ 0.3	e 0.3	(l, r) 0.4
u 0.1	r 0.2	r 0.2	r 0.2	l 0.2	$\{[(k, w), ?], u\}$ 0.3	
w 0.05	u 0.1	u 0.1	$[(k, w), ?]$ 0.2	r 0.2		
e 0.3	? 0.1	? 0.1	u 0.1			

r 0.2	k 0.05	(k,w) 0.1				
? 0.1	w 0.05					

Symbol	**Probability**	**Codeword**
k	0.05	10101
l	0.2	01
u	0.1	100
w	0.05	10100
e	0.3	11
r	0.2	00
?	0.1	1011

Figure 2.1: An example of Huffman codeword construction

In this example, the average codeword length is 2.6 bits per symbol. In general, the average codeword length is defined as

$$l_{avg} = \sum l_i p_i \tag{2.1}$$

where l_i is the codeword length (in bits) for the codeword corresponding to symbol s_i. The average codeword length is a measure of the compression ratio. Since our alphabet has seven symbols, a fixed-length coder would require at least three bits per codeword. In this example, we have reduced the representation from three bits per symbol to 2.6 bits per symbol; thus, the corresponding compression ratio can be stated as 3/2.6=1.15. For the lossless compression of typical image or video data, compression ratios in excess of two are hard to come by.

Properties of Huffman Codes

According to Shannon, the entropy of a source S is defined as

$$H(s) = n = \sum p_i \log_2 (1/p_i) \tag{2.2}$$

where, as before, p_i denotes the probability that symbol s_i from S will occur. From information theory, if the symbols are distinct, then the average number of bits needed to encode them is always bounded from below by their entropy. For example, for the alphabet used in the previous section, the average length is bounded by 2.6 bits per symbol. It can be shown that Huffman codewords satisfy the constraints $n \le l_{avg} < n+1$; that is, the average length is very close to the optimum. A tighter bound is

$n \le l_{avg} < p + 0.086$, where p is the probability of the most frequently occurring symbol**. The equality is achieved when all symbol probabilities are inverse powers of two**.

The Huffman code table construction process, as was described here, is referred to as a **bottom-up** method, since we perform the contraction process on the two least frequently occurring symbols. In recent years, **top-down** construction methods have also been published in the literature.

The code construction process has a complexity of $O(N \log_2 N)$. With presorting of the input symbol probabilities, code construction methods with complexity $O(N)$ are presently known.

In the example, one can observe that no codeword is a **prefix** for another codeword. Such a code is referred to as a **prefix-condition** code. Huffman codes satisfy always the prefix-condition.

Due to the prefix-condition property, Huffman codes are **uniquely decodable**. Not every uniquely decodable code satisfies the prefix-condition. A code such as 0, 01, 011, 0111 does not satisfy the prefix-condition, since zero is a prefix for all of the codewords; however, every codeword is uniquely decodable, since a zero signifies the start of a new codeword.

If we have a binary representation for the codewords, the complement of this representation is also a valid set of Huffman codewords. The choice of using the codeword set or the corresponding complement set depends on the application. For instance, if the Huffman codewords are to be transmitted over a noisy channel where the probability of error of a one being received as a zero is higher than the probability of error of a zero being received as a one, then one would choose the codeword set for which the bit zero has a higher probability of occurrence. This will improve the performance of the Huffman coder in this noisy channel.

In Huffman coding, fixed-length input symbols are mapped into variable-length codewords. Since there are no fixed-size boundaries between codewords, if some of the bits in the compressed stream are received incorrectly or if they are not received at all due to dropouts, all the data are lost. This potential loss can be prevented by using special markers within the compressed bit stream to designate the start or end of a compressed stream packet.

Extended Huffman Codes

Suppose we have three symbols with probabilities as shown in the following table. The Huffman codeword for each symbol is also shown.

Symbol	Probability	Code
s_1	0.8	0
s_2	0.02	11
s_3	0.18	10

For the above set of symbols we have:

Entropy $H(s) = n = 0.816$ bits/symbol.

Average number of bits per symbol $l_{avg} = 1.2$ bits/symbol.

Redundancy $l_{avg} - n = 1.2 - 0.816 = 0.384$ or $\frac{l_{avg} - n}{n}\% = 47\%$ of entropy.

For this particular example Huffman code gives a poor compression. This is because one of the symbols (s_1) has significantly higher probability of occurrence compared to the others. Suppose we merge the symbols in groups of two symbols. In the next table the extended alphabet and corresponding probabilities and Huffman codewords are shown.

Symbol	Probability	Code
s_1s_1	0.64	0
s_1s_2	0.016	10101
s_1s_3	0.144	11
s_2s_1	0.016	101000
s_2s_2	0.0004	10100101
s_2s_3	0.0036	1010011
s_3s_1	0.144	100
s_3s_2	0.0036	10100100
s_3s_3	0.0324	1011

Table: The extended alphabet and corresponding Huffman code

For the new extended alphabet we have

$l_{avg} = 1.7516$ bits/**new symbol** or $l_{avg} = 0.8758$ bits/original symbol.

Redundancy $l_{avg} - n = 0.8758 – 0.816 = 0.0598$ or $\frac{l_{avg} - n}{n}\% = 7\%$ of entropy.

We see that by coding the extended alphabet a significantly better compression is achieved. The above process is called **Extended Huffman Coding**.

Main Limitations of Huffman Coding

- To achieve the entropy of a DMS (Discrete Memoryless SourCe), the symbol probabilities should be negative powers of 2 (i.e. $\log p_i$ is an integer).
- Can not assign fractional codelengths.
- Can not efficiently adapt to changing source statistics.
- To improve coding efficiency $H(s)/l_{avg}$ we can encode the symbols of an extended source. However number of entries in Huffman table grows exponentially with block size.
  There are also cases where even the extended Huffman does not work. Suppose we have the following case:

Symbol	Probability	Code
s_1	0.95	0
s_2	0.02	11
s_3	0.03	10

Table: Huffman code for three symbol alphabet

Entropy $H(s) = n = 0.335$ bits/symbol.

Average number of bits per symbol $l_{avg} = 1.05$ bits/symbol.

Redundancy $l_{avg} - n = 1.05 - 0.335 = 0.715$ or $\frac{l_{avg} - n}{n}\% = 213\%$ of entropy!

Suppose we merge the symbols in groups of two symbols. In the next table the extended alphabet and corresponding probabilities and Huffman codewords are shown.

Symbol	Probability	Code
$s_1 s_1$	0.9025	0
$s_1 s_2$	0.019	111
$s_1 s_3$	0.0285	100
$s_2 s_1$	0.019	1101
$s_2 s_2$	0.0004	110011
$s_2 s_3$	0.0006	110001
$s_3 s_1$	0.0285	101
$s_3 s_2$	0.0006	110010
$s_3 s_3$	0.0009	110000

Table: The extended alphabet and corresponding Huffman code

For the new extended alphabet we have

$l_{avg} = 1.222$ bits/**new symbol** or $l_{avg} = 0.611$ bits/original symbol.

Redundancy $\frac{l_{avg} - n}{n}\% = 72\%$ of entropy.

For this example it is proven that redundancy drops to acceptable values by merging the original symbols in groups of 8 symbols! and in that case the alphabet size is 6561 new symbols!

Arithmetic coding solves many limitations of Huffman coding. Arithmetic coding is out of the scope of this course.

3 HUFFMAN DECODING

The Huffman encoding process is relatively straightforward. The symbol-to-codeword mapping table provided by the modeler is used to generate the codewords for each input symbol. On the other hand, the Huffman decoding process is somewhat more complex.

Bit-Serial Decoding

Let us assume that the binary coding tree is also available to the decoder. In practice, this tree can be reconstructed from the symbol-to-codeword mapping table that is known to both the encoder and the decoder. The decoding process consists of the following steps:

1. Read the input compressed stream bit by bit and traverse the tree until a leaf node is reached.
2. As each bit in the input stream is used, it is discarded. When the leaf node is reached, the Huffman decoder outputs the symbol at the leaf node. This completes the decoding for this symbol.

We repeat these steps until all of the input is consumed. For the example discussed in the previous section, since the longest codeword is five bits and the shortest codeword is two bits, the decoding bit rate is not the same for all symbols. Hence, this scheme has a fixed input bit rate but a variable output symbol rate.

Lookup-Table-Based Decoding

Lookup-table-based methods yield a constant decoding symbol rate. The lookup table is constructed at the decoder from the symbol-to-codeword mapping table. If the longest codeword in this table is L bits, then a 2^L entry lookup table is needed. Recall the first example that we presented in that section where $L = 5$. Specifically, the lookup table construction for each symbol s_i is as follows:

- Let c_i be the codeword that corresponds to symbol s_i. Assume that c_i has l_i bits. We form an $L-$ bit address in which the first l_i bits are c_i and the remaining $L - l_i$ bits take on all possible combinations of zero and one. Thus, for the symbol s_i there will be 2^{L-l_i} addresses.
- At each entry we form the two-tuple (s_i, l_i).

Decoding using the lookup-table approach is relatively easy:

1. From the compressed input bit stream, we read in L bits into a buffer.
2. We use the $L-$bit word in the buffer as an address into the lookup table and obtain the corresponding symbol, say s_k. Let the codeword length be l_k. We have now decoded one symbol.
3. We discard the first l_k bits from the buffer and we append to the buffer, the next l_k bits from the input, so that the buffer has again L bits.
4. We repeat Steps 2 and 3 until all of the symbols have been decoded.

The primary advantages of lookup-table-based decoding are that it is fast and that the decoding rate is constant for all symbols, regardless of the corresponding codeword length. However, the input bit rate is now variable. For image or video data, the longest codeword could be around 16 to 20 bits. Thus, in some applications, the lookup table approach may be impractical due to space constraints.
Variants on the basic theme of lookup-table-based decoding include using hierarchical lookup tables and combinations of lookup table and bit-by-bit decoding.
There are codeword construction methods that facilitate lookup-table-based decoding by constraining the maximum codeword length to a fixed-size L, but these are out of the scope of this course.

4 STANDARDS FOR LOSSLESS COMPRESSION

Standards related to the coding and transmission of signals over public telecommunication channels are developed under the auspices of the telecommunication standardization sector of the International Telecommunication Union (ITU-T). This sector was formerly known as the CCITT. The first standards for lossless compression were developed for facsimile applications. Scanned images used in such applications are bitonal, that is, the pixels take on one of two values, black or white, and these values are represented with one bit per pixel.

Facsimile Compression Standards and Run-Length Coding Scheme

In every bitonal image there are large regions that are either all white or all black. For instance, in Figure 4.1, we show a few pixels of a line in a bitonal image. Note that, the six contiguous pixels of the same color can be described as a run of six pixels with value 0. Thus, if each pixel of the image is remapped from say, its (position, value) to a **run** and **value**, then a more compact description can be obtained. In our example, no more than four bits are needed to describe the six-pixel run. In general, for many document type images, significant compression can be achieved using such preprocessing. Such a mapping scheme is referred to as a run-length coding scheme.

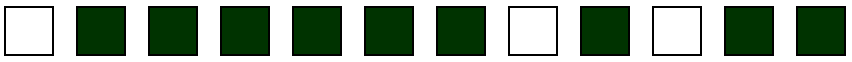


Figure 4.1: Sample scanline of a bitonal image

The combination of a run-length coding scheme followed by a Huffman coder forms the basis of the image coding standards for facsimile applications. These standards include the following:

- ITU-T Rec. T.4 (also known as Group 3). There are two coding approaches within this standard.

  1. Modified Huffman (MH) code. The image is treated as a sequence of scanlines, and a run-length description is first obtained for each line. A Huffman code is then applied to the (run, value) description. A separate Huffman code is used to distinguish between black and white runs, since the characteristics of these runs are quite different. The Huffman code table is static: that is, it does not change from image to image. For error-detection purposes, after each line is coded, an EOL (end of line) codeword is inserted.

  2. Modified Read (MR) code. Here, pixel values in a previous line are used as predictors for the current line. This is followed by a run-length description and a static Huffman code as in the MH code. An EOL codeword is also used. To prevent error propagation, MR coding is mixed with MH coding periodically.

- ITU-T Rec. T.6 (also known as Group 4). The coding technique used here is referred to as a Modified Modified Read (MMR) code. This code is a simplification of the MR code, wherein the error-protection mechanisms in the MR code are removed so as to improve the overall compression ratio.

These compression standards yield good compression (20:1 to 50:1) for business-type scanned documents. For images composed of natural scenes and rendered as bitonal images using a halftoning technique the compression ratio is severely degraded. In Figure 4.2, the image on the left possesses characteristics representative of business document images whereas the image on the right is a typical halftone image. The former is characterized by long runs of black or white, and the static Huffman code in the facsimile compression standards is matched to these run-lengths. In the latter image, the run-lengths are relatively short, spanning only one to two pixels and the static Huffman code is not matched to such runs. An adaptive arithmetic coder is better suited for such images.

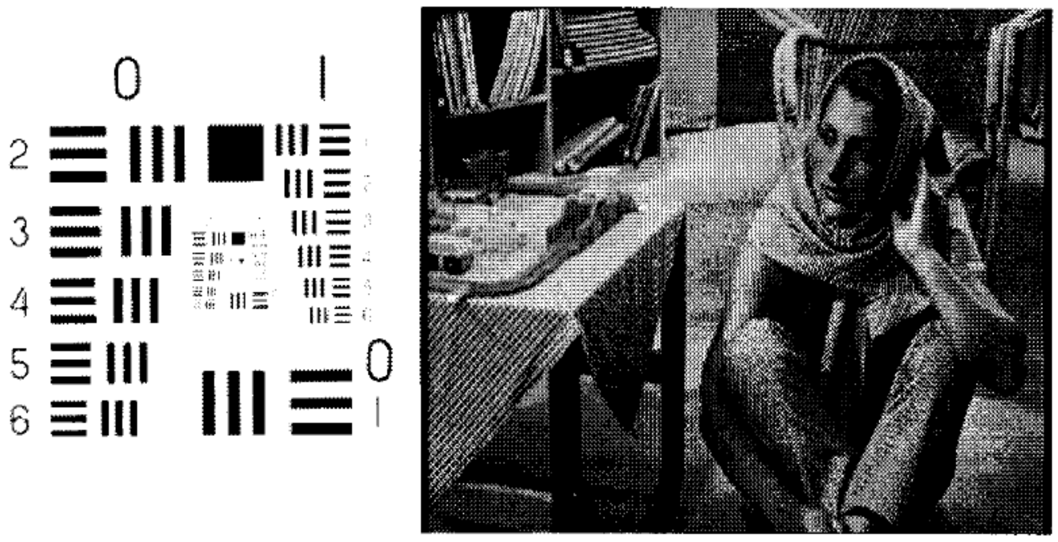


Figure 4.2: Typical bitonal images

The JBIG Compression Standard

Recently, a compression standard was developed to efficiently compress halftone as well as business document type images. This is the JBIG (Joint Binary Image Experts Group) compression standard. Its standard nomenclature is ISO/IEC IS 11544, ITU-T Rec. T.82. The JBIG compression standard consists of a modeler and an arithmetic coder. The modeler is used to estimate the symbol probabilities that are then used by the arithmetic coder. **The JBIG is out of the scope of this course.**

In Tables 4.1 below, we provide a simple complexity analysis between the JBIG coding scheme and the ITU-T Rec. T.6 facsimile compression standard. We also provide compression ratios for two typical images: a 202 Kbyte halftone image and a 1 Mbyte image, primarily comprised of text. The latter image is referred to as letter in the table. For business-type documents, JBIG yields 20 percent to 50 percent more compression than the facsimile compression standards ITU-T Rec. T.4 and Rec. T.6. For halftone images, compression ratios with JBIG are two to five times more than those obtained with the facsimile compression standards. However, software implementations of JBIG compression on a general purpose computer are two to three times slower than implementations of the ITU-T Rec. T.4 and T.6 standards.

The JBIG standard can also handle grayscale images by processing each bit-plane of a grayscale image as separate bitonal images.

	JBIG-Baselayer	**ITU-T Rec. T.6**
Complexity Parameters	Three-line template, AT-max=16	2-D runlength, Huffman code
Memory	1589 bytes	1024 bytes
Buffer	Three scanlines	Two scanlines
Operations	Add, shift	Add, shift, compare

Compression		
Halftone Image	5.2:1	1.5:1
Letter Image	48:1	33.3:1

Tables 4.1: Comparative analysis between JBIG and the ITU-T Rec. T.6 facsimile compression standards

The Lossless JPEG Standard

Most people know JPEG as a transform-based lossy compression standard. JPEG (Joint Photographic Experts Group), like JBIG, has been developed jointly by both the ITU-T and the ISO. We will describe this standard in greater detail in a subsequent section; however, here, we describe briefly the lossless mode of compression supported within this standard. The lossless compression method within JPEG is fully independent from transform-based coding. Instead, it uses differential coding to form prediction residuals that are then coded with either a Huffman coder or an arithmetic coder. As explained earlier, the prediction residuals usually have a lower entropy; thus, they are more amenable to compression than the original image pixels.

In lossless JPEG, one forms a prediction residual using previously encoded pixels in the current line and/or the previous line. The prediction residual for pixel X in Figure 4.3 is defined as $r = y - X$

where y can be any of the following functions:

$y = 0$

$y = a$

$y = b$

$y = c$

$y = a + b - c$

$y = a + (b - c)/2$

$y = b + (a - c)/2$

$y = (a + b)/2$

Note that, pixel values at pixel positions a, b, and c, are available to both the encoder and the decoder prior to processing X. The particular choice for the y function is defined in the scan header of the compressed stream so that both the encoder and the decoder use identical functions. Divisions by two are computed by performing a one-bit right shift.

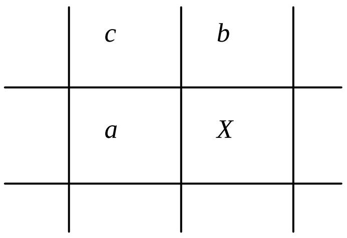


Figure 4.3: Lossless JPEG prediction kernel

The prediction residual is computed modulo 2. This residual is not directly Huffman coded. Instead, it is expressed as a pair of symbols: the category and the magnitude. The first symbol represents the number of bits needed to encode the magnitude. Only this value is Huffman coded. The magnitude categories for all possible values of the prediction residual are shown in Table 4.2. If, say, the prediction residual for X is 42, then from Table 4.2 we determine that this value belongs to category 6; that is, we need an additional six bits to uniquely determine the value 42. The prediction residual is then mapped into the two-tuple (6, 6-bit code for 42). Category 6 is Huffman coded, and the compressed representation for the prediction residual consists of this Huffman codeword followed by the 6-bit representation for the magnitude. In general, if the value of the residual is positive, then the code for the magnitude is its direct binary representation. If the residual is negative, then the code for the magnitude is the one's complement of its absolute value. Therefore, codewords for negative residual always start wish a zero bit.

Example 3: Consider Figure 4.3 with pixel values $a = 100$, $b = 191$, $c = 100$, and $X = 180$. Let

$y = (a+b)/2$; then $y = 145$, and the prediction residual is $r = 145 - 180 = -35$. From Table 4.2, -35 belongs to category 6. The binary number for 35 is 100011, and its one's complement is 011100. Thus, -35 is represented as (6,011100). If the Huffman code for six is 1110, then -35 is coded by the 10-bit codeword 1110011100. Without entropy coding, -35 would require 16 bits.

In the decoder, the category (that is, 6) is extracted first. Thus, the next six bits, 011100, correspond to the magnitude of the residual. Since the most significant bit is zero, the residual is negative. After taking the one's complement of 011100, the decoded value of the residual r is -35. The a and b bits have already been decoded; thus, $y = 145$ as before, and $X = y + 35 = 180$.

Category	**Prediction Residual**
0	0
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
12	-4095, ..., -2048, 2048, ..., 4095
13	-8191, ..., -4096, 4096, ..., 8191
14	-16383, …,-8192, 8192, ..., 16383
15	-32767, ..., -16384, 16384, ..., 32767
16	32768

Table 4.2: Prediction residual categories for lossless JPEG compression.

This notion of using a category table is a form of context modeling and simplifies the Huffman coder. Without categorization of the prediction residuals, we would require a Huffman table for an alphabet of 2^{16} symbols. Such a large codeword table would complicate both the codeword construction process and the decoding process.

Lossless JPEG outperforms JBIG for typical grayscale images with more than six bits per pixel. At six bits per pixel or below, JBIG yields better compression ratios than JPEG. For typical images, such as the grayscale version of the halftone image of Figure 4.2, compression ratios in excess of 1.5 to 1 are quite difficult to achieve. The standards committee is currently working on developing new lossless compression techniques that can outperform the simple single-prediction, single-Huffman table coding method currently used in the lossless JPEG compression standard.

5 TO PROBE FURTHER

We have reviewed some of the algorithms and standards for lossless compression. Huffman coding, in particular, is the most widely used entropy coder in the standards. A general discussion on entropy coders can be found in any textbook on information theory.

In many practical implementations, the maximum codeword length of a Huffman code needs to be constrained. Several approaches for constructing such a code have been developed, but are out of the scope of this course.

Huffman coding methods are amenable to simpler software and hardware implementations; however, techniques based on arithmetic coding tend to yield a higher compression ratio. Arithmetic coding is quite complicated, and therefore is also out of the scope of this course.

References

[1] ***Digital Image Processing*** by R. C. Gonzales and R. E. Woods, Addison-Wesley Publishing Company, 1992.

[2] ***Two-Dimensional Signal and Image Processing*** by J. S. Lim, Prentice Hall, 1990.