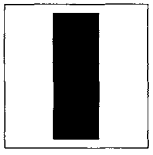
7

Lossless Image Compression

7.1 Overview



n this chapter we examine a number of schemes used for lossless compression of images. We will look at schemes for compression of grayscale and color images as well as schemes for compression of binary images. Among these schemes are several that are a part of international standards.

7.2 Introduction

In the previous chapters we have focused on compression techniques. Although some of them may apply to some preferred applications, the focus has been on the technique rather than on the application. However, there are certain techniques for which it is impossible to separate the technique from the application. This is because the techniques rely upon the properties or characteristics of the application. Therefore, we have several chapters in this book that focus on particular applications. In this chapter we will examine techniques specifically geared toward lossless image compression. Later chapters will examine speech, audio, and video compression.

In the previous chapters we have seen that a more skewed set of probabilities for the message being encoded results in better compression. In Chapter 6 we saw how the use of context to obtain a skewed set of probabilities can be especially effective when encoding text. We can also transform the sequence (in an invertible fashion) into another sequence that has the desired property in other ways. For example, consider the following sequence:

1	2	5	7	2	−2	0	−5	−3	−1	1	−2	−7	−4	−2	1	3	4

If we consider this sample to be fairly typical of the sequence, we can see that the probability of any given number being in the range from -7 to 7 is about the same. If we were to encode this sequence using a Huffman or arithmetic code, we would use almost 4 bits per symbol.

Instead of encoding this sequence directly, we could do the following: add two to the previous number in the sequence and send the difference between the current element in the sequence and this *predicted* value. The transmitted sequence would be

1	−1	1	0	−7	−4	0	−7	0	0	0	−5	−7	1	0	1	0	−1

This method uses a rule (add two) and the history (value of the previous symbol) to generate the new sequence. If the rule by which this *residual sequence* was generated is known to the decoder, it can recover the original sequence from the residual sequence. The length of the residual sequence is the same as the original sequence. However, notice that the residual sequence is much more likely to contain 0s, 1s, and -1s than other values. That is, the probability of 0, 1, and -1 will be significantly higher than the probabilities of other numbers. This, in turn, means that the entropy of the residual sequence will be low and, therefore, provide more compression.

We used a particular method of prediction in this example (add two to the previous element of the sequence) that was specific to this sequence. In order to get the best possible performance, we need to find the prediction approach that is best suited to the particular data we are dealing with. We will look at several prediction schemes used for lossless image compression in the following sections.

7.2.1 The Old JPEG Standard

The Joint Photographic Experts Group (JPEG) is a joint ISO/ITU committee responsible for developing standards for continuous-tone still-picture coding. The more famous standard produced by this group is the lossy image compression standard. However, at the time of the creation of the famous JPEG standard, the committee also created a lossless standard [73]. At this time the standard is more or less obsolete, having been overtaken by the much more efficient JPEG-LS standard described later in this chapter. However, the old JPEG standard is still useful as a first step into examining predictive coding in images.

The old JPEG lossless still compression standard [73] provides eight different predictive schemes from which the user can select. The first scheme makes no prediction. The next seven are listed below. Three of the seven are one-dimensional predictors, and four are two-dimensional prediction schemes. Here, $I(i, j)$ is the (i, j)th pixel of the original image, and $\hat{I}(i, j)$ is the predicted value for the (i, j)th pixel.

$$\mathbf{1}\quad \hat{I}(i,j) = I(i-1,j) \tag{7.1}$$

$$\mathbf{2}\quad \hat{I}(i,j) = I(i,j-1) \tag{7.2}$$

$$\mathbf{3}\quad \hat{I}(i,j) = I(i-1,j-1) \tag{7.3}$$

$$\mathbf{4}\quad \hat{I}(i,j) = I(i,j-1) + I(i-1,j) - I(i-1,j-1) \tag{7.4}$$

$$5 \quad \hat{I}(i,j) = I(i,j-1) + (I(i-1,j) - I(i-1,j-1))/2 \tag{7.5}$$

$$6 \quad \hat{I}(i,j) = I(i-1,j) + (I(i,j-1) - I(i-1,j-1))/2 \tag{7.6}$$

$$7 \quad \hat{I}(i,j) = (I(i,j-1) + I(i-1,j))/2 \tag{7.7}$$

Different images can have different structures that can be best exploited by one of these eight modes of prediction. If compression is performed in a nonreal-time environment—for example, for the purposes of archiving—all eight modes of prediction can be tried and the one that gives the most compression is used. The mode used to perform the prediction can be stored in a 3-bit header along with the compressed file. We encoded our four test images using the various JPEG modes. The residual images were encoded using adaptive arithmetic coding. The results are shown in Table 7.1.

The best results—that is, the smallest compressed file sizes—are indicated in bold in the table. From these results we can see that a different JPEG predictor is the best for the different images. In Table 7.2, we compare the best JPEG results with the file sizes obtained using GIF and PNG. Note that PNG also uses predictive coding with four possible predictors, where each row of the image can be encoded using a different predictor. The PNG approach is described in Chapter 5.

Even if we take into account the overhead associated with GIF, from this comparison we can see that the predictive approaches are generally better suited to lossless image compression than the dictionary-based approach when the images are "natural" gray-scale images. The situation is different when the images are graphic images or pseudocolor images. A possible exception could be the Earth image. The best compressed file size using the second JPEG mode and adaptive arithmetic coding is 32,137 bytes, compared to 34,276 bytes using GIF. The difference between the file sizes is not significant. We can see the reason by looking at the Earth image. Note that a significant portion of the image is the

TABLE 7.1 Compressed file size in bytes of the residual images obtained using the various JPEG prediction modes.

Image	JPEG 0	JPEG 1	JPEG 2	JPEG 3	JPEG 4	JPEG 5	JPEG 6	JPEG 7
Sena	53,431	37,220	31,559	38,261	31,055	**29,742**	33,063	32,179
Sensin	58,306	41,298	37,126	43,445	**32,429**	33,463	35,965	36,428
Earth	38,248	32,295	**32,137**	34,089	33,570	33,057	33,072	32,672
Omaha	56,061	**48,818**	51,283	53,909	53,771	53,520	52,542	52,189

TABLE 7.2 Comparison of the file sizes obtained using JPEG lossless compression, GIF, and PNG.

Image	Best JPEG	GIF	PNG
Sena	31,055	51,085	31,577
Sensin	32,429	60,649	34,488
Earth	32,137	34,276	26,995
Omaha	48,818	61,341	50,185

background, which is of a constant value. In dictionary coding, this would result in some very long entries that would provide significant compression. We can see that if the ratio of background to foreground were just a little different in this image, the dictionary method in GIF might have outperformed the JPEG approach. The PNG approach which allows the use of a different predictor (or no predictor) on each row, prior to dictionary coding significantly outperforms both GIF and JPEG on this image.

7.3 CALIC

The Context Adaptive Lossless Image Compression (CALIC) scheme, which came into being in response to a call for proposal for a new lossless image compression scheme in 1994 [74, 75], uses both context and prediction of the pixel values. The CALIC scheme actually functions in two modes, one for gray-scale images and another for bi-level images. In this section, we will concentrate on the compression of gray-scale images.

In an image, a given pixel generally has a value close to one of its neighbors. Which neighbor has the closest value depends on the local structure of the image. Depending on whether there is a horizontal or vertical edge in the neighborhood of the pixel being encoded, the pixel above, or the pixel to the left, or some weighted average of neighboring pixels may give the best prediction. How close the prediction is to the pixel being encoded depends on the surrounding texture. In a region of the image with a great deal of variability, the prediction is likely to be further from the pixel being encoded than in the regions with less variability.

In order to take into account all these factors, the algorithm has to make a determination of the environment of the pixel to be encoded. The only information that can be used to make this determination has to be available to both encoder and decoder.

Let's take up the question of the presence of vertical or horizontal edges in the neighborhood of the pixel being encoded. To help our discussion, we will refer to Figure 7.1. In this figure, the pixel to be encoded has been marked with an X. The pixel above is called the north pixel, the pixel to the left is the west pixel, and so on. Note that when pixel X is being encoded, all other marked pixels (N, W, NW, NE, WW, NN, NE, and NNE) are available to both encoder and decoder.

		NN	*NNE*
	NW	*N*	*NE*
WW	*W*	*X*	

FIGURE 7.1 Labeling the neighbors of pixel X.

We can get an idea of what kinds of boundaries may or may not be in the neighborhood of X by computing

$$d_h = |W - WW| + |N - NW| + |NE - N|$$
$$d_v = |W - NW| + |N - NN| + |NE - NNE|.$$

The relative values of d_h and d_v are used to obtain the initial prediction of the pixel X. This initial prediction is then refined by taking other factors into account. If the value of d_h is much higher than the value of d_v, this will mean there is a large amount of horizontal variation, and it would be better to pick N to be the initial prediction. If, on the other hand, d_v is much larger than d_h, this would mean that there is a large amount of vertical variation, and the initial prediction is taken to be W. If the differences are more moderate or smaller, the predicted value is a weighted average of the neighboring pixels.

The exact algorithm used by CALIC to form the initial prediction is given by the following pseudocode:

```
if d_h - d_v > 80
  X̂ ← N
else if d_v - d_h > 80
  X̂ ← W
else
{
      X̂ ← (N + W)/2 + (NE - NW)/4
      if d_h - d_v > 32
        X̂ ← (X̂ + N)/2
      else if d_v - d_h > 32
        X̂ ← (X̂ + W)/2
      else if d_h - d_v > 8
        X̂ ← (3X̂ + N)/4
      else if d_v - d_h > 8
        X̂ ← (3X̂ + W)/4
}
```

Using the information about whether the pixel values are changing by large or small amounts in the vertical or horizontal direction in the neighborhood of the pixel being encoded provides a good initial prediction. In order to refine this prediction, we need some information about the interrelationships of the pixels in the neighborhood. Using this information, we can generate an offset or refinement to our initial prediction. We quantify the information about the neighborhood by first forming the vector

$$[N, W, NW, NE, NN, WW, 2N - NN, 2W - WW]$$

We then compare each component of this vector with our initial prediction $\hat{X}$. If the value of the component is less than the prediction, we replace the value with a 1; otherwise

we replace it with a 0. Thus, we end up with an eight-component binary vector. If each component of the binary vector was independent, we would end up with 256 possible vectors. However, because of the dependence of various components, we actually have 144 possible configurations. We also compute a quantity that incorporates the vertical and horizontal variations and the previous error in prediction by

$$\delta = d_h + d_v + 2|N - \hat{N}| \tag{7.8}$$

where $\hat{N}$ is the predicted value of N. This range of values of δ is divided into four intervals, each being represented by 2 bits. These four possibilities, along with the 144 texture descriptors, create $144 \times 4 = 576$ contexts for X. As the encoding proceeds, we keep track of how much prediction error is generated in each context and offset our initial prediction by that amount. This results in the final predicted value.

Once the prediction is obtained, the difference between the pixel value and the prediction (the prediction error, or residual) has to be encoded. While the prediction process outlined above removes a lot of the structure that was in the original sequence, there is still some structure left in the residual sequence. We can take advantage of some of this structure by coding the residual in terms of its context. The context of the residual is taken to be the value of δ defined in Equation (7.8). In order to reduce the complexity of the encoding, rather than using the actual value as the context, CALIC uses the range of values in which δ lies as the context. Thus:

$$0 \le \delta < q_1 \Rightarrow \text{Context 1}$$
$$q_1 \le \delta < q_2 \Rightarrow \text{Context 2}$$
$$q_2 \le \delta < q_3 \Rightarrow \text{Context 3}$$
$$q_3 \le \delta < q_4 \Rightarrow \text{Context 4}$$
$$q_4 \le \delta < q_5 \Rightarrow \text{Context 5}$$
$$q_5 \le \delta < q_6 \Rightarrow \text{Context 6}$$
$$q_6 \le \delta < q_7 \Rightarrow \text{Context 7}$$
$$q_7 \le \delta < q_8 \Rightarrow \text{Context 8}$$

The values of q_1–q_8 can be prescribed by the user.

If the original pixel values lie between 0 and $M-1$, the differences or prediction residuals will lie between $-(M-1)$ and $M-1$. Even though most of the differences will have a magnitude close to zero, for arithmetic coding we still have to assign a count to all possible symbols. This means a reduction in the size of the intervals assigned to values that do occur, which in turn means using a larger number of bits to represent these values. The CALIC algorithm attempts to resolve this problem in a number of ways. Let's describe these using an example.

Consider the sequence

$$x_n : 0,\ 7,\ 4,\ 3,\ 5,\ 2,\ 1,\ 7$$

We can see that all the numbers lie between 0 and 7, a range of values that would require 3 bits to represent. Now suppose we predict a sequence element by the previous element in the sequence. The sequence of differences

$$r_n = x_n - x_{n-1}$$

is given by

$$r_n : 0,\ 7,\ -3,\ -1,\ 2,\ -3,\ -1,\ 6$$

If we were given this sequence, we could easily recover the original sequence by using

$$x_n = x_{n-1} + r_n.$$

However, the prediction residual values r_n lie in the $[-7, 7]$ range. That is, the alphabet required to represent these values is almost twice the size of the original alphabet. However, if we look closely we can see that the value of r_n actually lies between $-x_{n-1}$ and $7 - x_{n-1}$. The smallest value that r_n can take on occurs when x_n has a value of 0, in which case r_n will have a value of $-x_{n-1}$. The largest value that r_n can take on occurs when x_n is 7, in which case r_n has a value of $7 - x_{n-1}$. In other words, given a particular value for x_{n-1}, the number of different values that r_n can take on is the same as the number of values that x_n can take on. Generalizing from this, we can see that if a pixel takes on values between 0 and $M-1$, then given a predicted value $\hat{X}$, the difference $X - \hat{X}$ will take on values in the range $-\hat{X}$ to $M - 1 - \hat{X}$. We can use this fact to map the difference values into the range $[0, M-1]$, using the following mapping:

$$\begin{aligned}
0 &\rightarrow 0\\
1 &\rightarrow 1\\
-1 &\rightarrow 2\\
2 &\rightarrow 3\\
&\vdots \quad \vdots\\
-\hat{X} &\rightarrow 2\hat{X}\\
\hat{X}+1 &\rightarrow 2\hat{X}+1\\
\hat{X}+2 &\rightarrow 2\hat{X}+2\\
&\vdots \quad \vdots\\
M-1-\hat{X} &\rightarrow M-1
\end{aligned}$$

where we have assumed that $\hat{X} \leq (M-1)/2$.

Another approach used by CALIC to reduce the size of its alphabet is to use a modification of a technique called *recursive indexing* [76]. Recursive indexing is a technique for representing a large range of numbers using only a small set. It is easiest to explain using an example. Suppose we want to represent positive integers using only the integers between 0 and 7—that is, a representation alphabet of size 8. Recursive indexing works as follows: If the number to be represented lies between 0 and 6, we simply represent it by that number. If the number to be represented is greater than or equal to 7, we first send the number 7, subtract 7 from the original number, and repeat the process. We keep repeating the process until the remainder is a number between 0 and 6. Thus, for example, 9 would be represented by 7 followed by a 2, and 17 would be represented by two 7s followed by a 3. The decoder, when it sees a number between 0 and 6, would decode it at its face value, and when it saw 7, would keep accumulating the values until a value between 0 and 6 was received. This method of representation followed by entropy coding has been shown to be optimal for sequences that follow a geometric distribution [77].

In CALIC, the representation alphabet is different for different coding contexts. For each coding context k, we use an alphabet $A_k = \{0, 1, \ldots, N_k\}$. Furthermore, if the residual occurs in context k, then the first number that is transmitted is coded with respect to context k; if further recursion is needed, we use the $k+1$ context.

We can summarize the CALIC algorithm as follows:

1. Find initial prediction $\hat{X}$.
2. Compute prediction context.
3. Refine prediction by removing the estimate of the bias in that context.
4. Update bias estimate.
5. Obtain the residual and remap it so the residual values lie between 0 and $M-1$, where M is the size of the initial alphabet.
6. Find the coding context k.
7. Code the residual using the coding context.

All these components working together have kept CALIC as the state of the art in lossless image compression. However, we can get almost as good a performance if we simplify some of the more involved aspects of CALIC. We study such a scheme in the next section.

7.4 JPEG-LS

The JPEG-LS standard looks more like CALIC than the old JPEG standard. When the initial proposals for the new lossless compression standard were compared, CALIC was rated first in six of the seven categories of images tested. Motivated by some aspects of CALIC, a team from Hewlett-Packard proposed a much simpler predictive coder, under the name LOCO-I (for low complexity), that still performed close to CALIC [78].

As in CALIC, the standard has both a lossless and a lossy mode. We will not describe the lossy coding procedures.

The initial prediction is obtained using the following algorithm:

```
if NW ≥ max(W, N)
X̂ = max(W, N)
else
{
    if NW ≤ min(W, N)
    X̂ = min(W, N)
    else
    X̂ = W + N − NW
}
```

This prediction approach is a variation of Median Adaptive Prediction [79], in which the predicted value is the median of the N, W, and NW pixels. The initial prediction is then refined using the average value of the prediction error in that particular context.

The contexts in JPEG-LS also reflect the local variations in pixel values. However, they are computed differently from CALIC. First, measures of differences D_1, D_2, and D_3 are computed as follows:

$$\begin{aligned} D_1 &= NE - N \\ D_2 &= N - NW \\ D_3 &= NW - W. \end{aligned}$$

The values of these differences define a three-component context vector $\mathbf{Q}$. The components of $\mathbf{Q}$ (Q_1, Q_2, and Q_3) are defined by the following mappings:

$$\begin{aligned} D_i \le -T_3 &\Rightarrow Q_i = -4 \\ -T_3 < D_i \le -T_2 &\Rightarrow Q_i = -3 \\ -T_2 < D_i \le -T_1 &\Rightarrow Q_i = -2 \\ -T_1 < D_i \le 0 &\Rightarrow Q_i = -1 \\ D_i = 0 &\Rightarrow Q_i = 0 \\ 0 < D_i \le T_1 &\Rightarrow Q_i = 1 \\ T_1 < D_i \le T_2 &\Rightarrow Q_i = 2 \\ T_2 < D_i \le T_3 &\Rightarrow Q_i = 3 \\ T_3 < D_i &\Rightarrow Q_i = 4 \end{aligned} \tag{7.9}$$

where T_1, T_2, and T_3 are positive coefficients that can be defined by the user. Given nine possible values for each component of the context vector, this results in $9 \times 9 \times 9 = 729$ possible contexts. In order to simplify the coding process, the number of contexts is reduced by replacing any context vector $\mathbf{Q}$ whose first nonzero element is negative by $-\mathbf{Q}$. Whenever

TABLE 7.3 Comparison of the file sizes obtained using new and old JPEG lossless compression standard and CALIC.

Image	Old JPEG	New JPEG	CALIC
Sena	31,055	27,339	26,433
Sensin	32,429	30,344	29,213
Earth	32,137	26,088	25,280
Omaha	48,818	50,765	48,249

this happens, a variable *SIGN* is also set to -1; otherwise, it is set to $+1$. This reduces the number of contexts to 365. The vector $\mathbf{Q}$ is then mapped into a number between 0 and 364. (The standard does not specify the particular mapping to use.)

The variable *SIGN* is used in the prediction refinement step. The correction is first multiplied by *SIGN* and then added to the initial prediction.

The prediction error r_n is mapped into an interval that is the same size as the range occupied by the original pixel values. The mapping used in JPEG-LS is as follows:

$$r_n < -\frac{M}{2} \Rightarrow r_n \leftarrow r_n + M$$

$$r_n > \frac{M}{2} \Rightarrow r_n \leftarrow r_n - M$$

Finally, the prediction errors are encoded using adaptively selected codes based on Golomb codes, which have also been shown to be optimal for sequences with a geometric distribution. In Table 7.3 we compare the performance of the old and new JPEG standards and CALIC. The results for the new JPEG scheme were obtained using a software implementation courtesy of HP.

We can see that for most of the images the new JPEG standard performs very close to CALIC and outperforms the old standard by 6% to 18%. The only case where the performance is not as good is for the Omaha image. While the performance improvement in these examples may not be very impressive, we should keep in mind that for the old JPEG we are picking the best result out of eight. In practice, this would mean trying all eight JPEG predictors and picking the best. On the other hand, both CALIC and the new JPEG standard are single-pass algorithms. Furthermore, because of the ability of both CALIC and the new standard to function in multiple modes, both perform very well on compound documents, which may contain images along with text.

7.5 Multiresolution Approaches

Our final predictive image compression scheme is perhaps not as competitive as the other schemes. However, it is an interesting algorithm because it approaches the problem from a slightly different point of view.

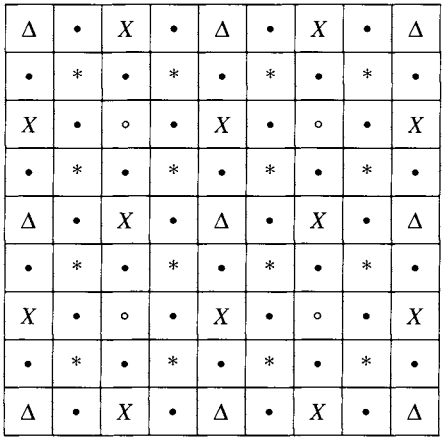


FIGURE 7.2 The HINT scheme for hierarchical prediction.

Multiresolution models generate representations of an image with varying spatial resolution. This usually results in a pyramidlike representation of the image, with each layer of the pyramid serving as a prediction model for the layer immediately below.

One of the more popular of these techniques is known as HINT (Hierarchical INTerpolation) [80]. The specific steps involved in HINT are as follows. First, residuals corresponding to the pixels labeled Δ in Figure 7.2 are obtained using linear prediction and transmitted. Then, the intermediate pixels (∘) are estimated by linear interpolation, and the error in estimation is then transmitted. Then, the pixels X are estimated from Δ and ∘, and the estimation error is transmitted. Finally, the pixels labeled $*$ and then • are estimated from known neighbors, and the errors are transmitted. The reconstruction process proceeds in a similar manner.

One use of a multiresolution approach is in progressive image transmission. We describe this application in the next section.

7.5.1 Progressive Image Transmission

The last few years have seen a very rapid increase in the amount of information stored as images, especially remotely sensed images (such as images from weather and other satellites) and medical images (such as CAT scans, magnetic resonance images, and mammograms). It is not enough to have information. We also need to make these images accessible to individuals who can make use of them. There are many issues involved with making large amounts of information accessible to a large number of people. In this section we will look at one particular issue—transmitting these images to remote users. (For a more general look at the problem of managing large amounts of information, see [81].)

Suppose a user wants to browse through a number of images in a remote database. The user is connected to the database via a 56 kbits per second (kbps) modem. Suppose the

images are of size 1024×1024, and on the average users have to look through 30 images before finding the image they are looking for. If these images were monochrome with 8 bits per pixel, this process would take close to an hour and 15 minutes, which is not very practical. Even if we compressed these images before transmission, lossless compression on average gives us about a two-to-one compression. This would only cut the transmission in half, which still makes the approach cumbersome. A better alternative is to send an approximation of each image first, which does not require too many bits but still is sufficiently accurate to give users an idea of what the image looks like. If users find the image to be of interest, they can request a further refinement of the approximation, or the complete image. This approach is called *progressive image transmission.*

Example 7.5.1:

A simple progressive transmission scheme is to divide the image into blocks and then send a representative pixel for the block. The receiver replaces each pixel in the block with the representative value. In this example, the representative value is the value of the pixel in the top-left corner. Depending on the size of the block, the amount of data that would need to be transmitted could be substantially reduced. For example, to transmit a 1024×1024 image at 8 bits per pixel over a 56 kbps line takes about two and a half minutes. Using a block size of 8×8, and using the top-left pixel in each block as the representative value, means we approximate the 1024×1024 image with a 128×128 subsampled image. Using 8 bits per pixel and a 56 kbps line, the time required to transmit this approximation to the image takes less than two and a half seconds. Assuming that this approximation was sufficient to let the user decide whether a particular image was the desired image, the time required now to look through 30 images becomes a minute and a half instead of the hour and a half mentioned earlier. If the approximation using a block size of 8×8 does not provide enough resolution to make a decision, the user can ask for a refinement. The transmitter can then divide the 8×8 block into four 4×4 blocks. The pixel at the upper-left corner of the upper-left block was already transmitted as the representative pixel for the 8×8 block, so we need to send three more pixels for the other three 4×4 blocks. This takes about seven seconds, so even if the user had to request a finer approximation every third image, this would only increase the total search time by a little more than a minute. To see what these approximations look like, we have taken the Sena image and encoded it using different block sizes. The results are shown in Figure 7.3. The lowest-resolution image, shown in the top left, is a 32×32 image. The top-left image is a 64×64 image. The bottom-left image is a 128×128 image, and the bottom-right image is the 256×256 original.

Notice that even with a block size of 8 the image is clearly recognizable as a person. Therefore, if the user was looking for a house, they would probably skip over this image after seeing the first approximation. If the user was looking for a picture of a person, they could still make decisions based on the second approximation.

Finally, when an image is built line by line, the eye tends to follow the scan line. With the progressive transmission approach, the user gets a more global view of the image very early in the image formation process. Consider the images in Figure 7.4. The images on the left are the 8×8, 4×4, and 2×2 approximations of the Sena image. On the right, we show

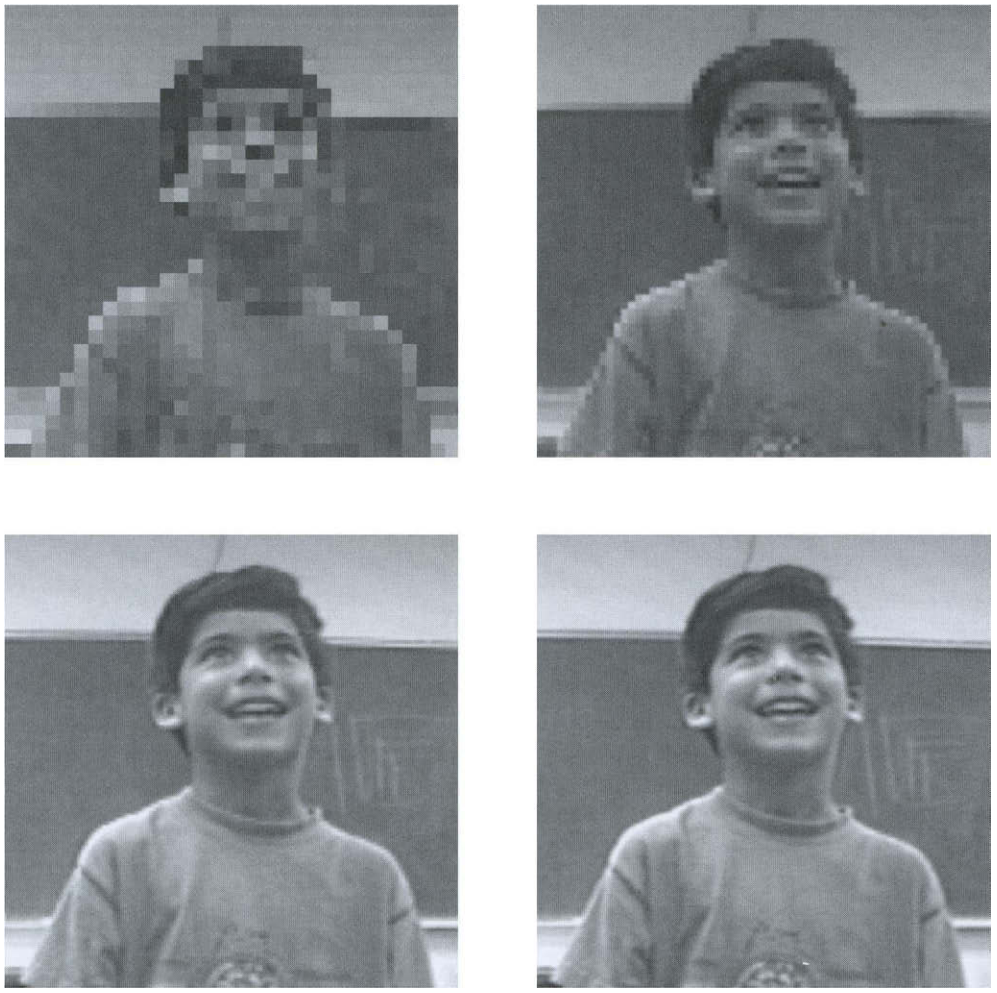


FIGURE 7. 3 **Sena image coded using different block sizes for progressive transmission. Top row: block size 8 x 8 and block size 4 x 4. Bottom row: block size 2 x 2 and original image.**

how much of the image we would see in the same amount of time if we used the standard line-by-line raster scan order. ♦

We would like the first approximations that we transmit to use as few bits as possible yet be accurate enough to allow the user to make a decision to accept or reject the image with a certain degree of confidence. As these approximations are lossy, many progressive transmission schemes use well-known lossy compression schemes in the first pass.

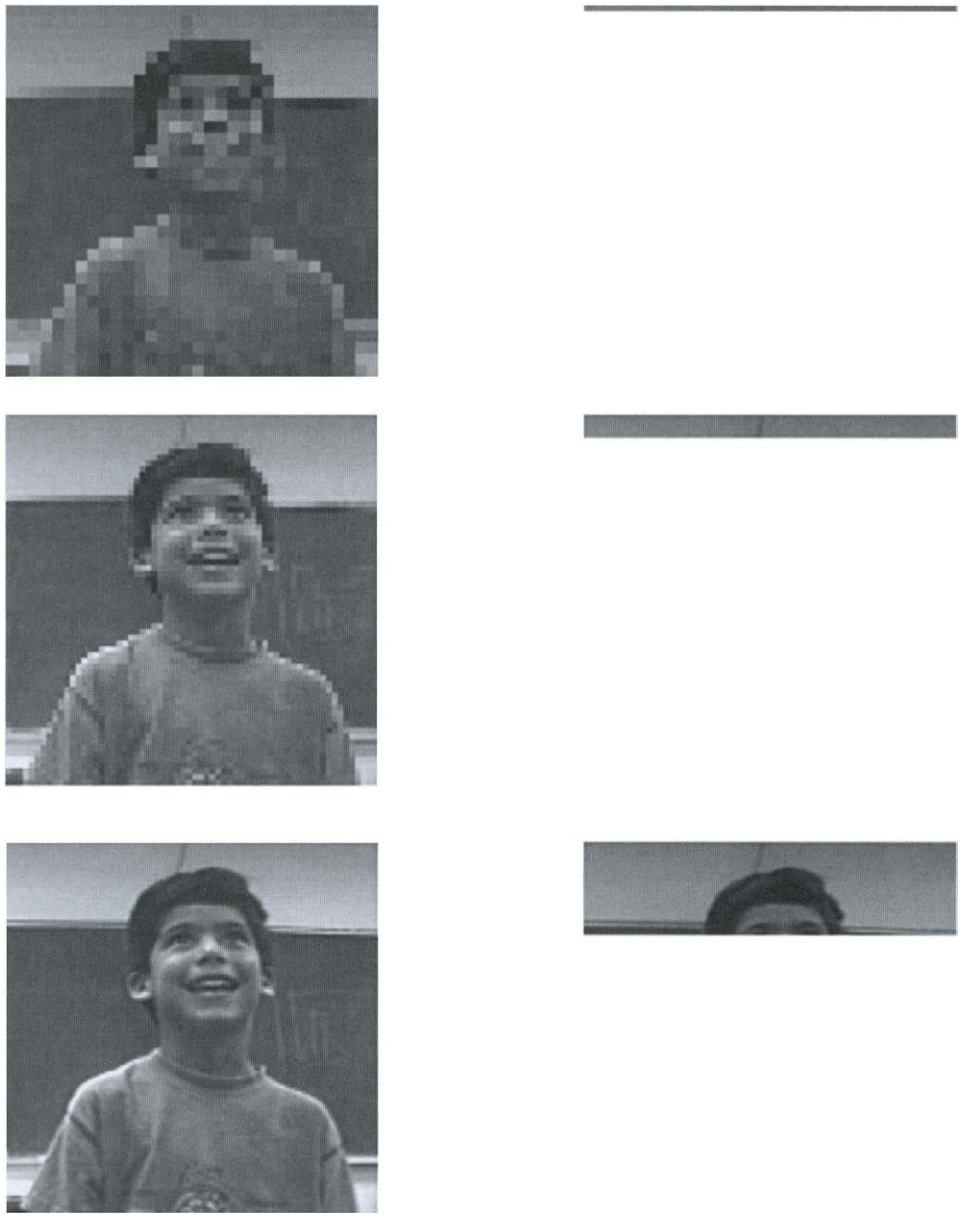


FIGURE 7.4 **Comparison between the received image using progressive transmission and using the standard raster scan order.**

The more popular lossy compression schemes, such as transform coding, tend to require a significant amount of computation. As the decoders for most progressive transmission schemes have to function on a wide variety of platforms, they are generally implemented in software and need to be simple and fast. This requirement has led to the development of a number of progressive transmission schemes that do not use lossy compression schemes for their initial approximations. Most of these schemes have a form similar to the one described in Example 7.5.1, and they are generally referred to as *pyramid schemes* because of the manner in which the approximations are generated and the image is reconstructed.

When we use the pyramid form, we still have a number of ways to generate the approximations. One of the problems with the simple approach described in Example 7.5.1 is that if the pixel values vary a lot within a block, the "representative" value may not be very representative. To prevent this from happening, we could represent the block by some sort of an average or composite value. For example, suppose we start out with a 512×512 image. We first divide the image into 2×2 blocks and compute the integer value of the average of each block [82, 83]. The integer values of the averages would constitute the penultimate approximation. The approximation to be transmitted prior to that can be obtained by taking the average of 2×2 averages and so on, as shown in Figure 7.5.

Using the simple technique in Example 7.5.1, we ended up transmitting the same number of values as the original number of pixels. However, when we use the mean of the pixels as our approximation, after we have transmitted the mean values at each level, we still have to transmit the actual pixel values. The reason is that when we take the integer part of the average we end up throwing away information that cannot be retrieved. To avoid this problem of data expansion, we can transmit the sum of the values in the 2×2 block. Then we only need to transmit three more values to recover the original four values. With this approach, although we would be transmitting the same number of values as the number of pixels in the image, we might still end up sending more bits because representing all possible

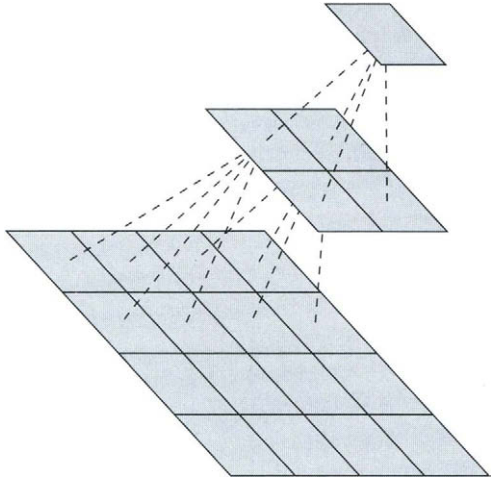


FIGURE 7.5 **The pyramid structure for progressive transmission.**

values of the sum would require transmitting 2 more bits than was required for the original value. For example, if the pixels in the image can take on values between 0 and 255, which can be represented by 8 bits, their sum will take on values between 0 and 1024, which would require 10 bits. If we are allowed to use entropy coding, we can remove the problem of data expansion by using the fact that the neighboring values in each approximation are heavily correlated, as are values in different levels of the pyramid. This means that differences between these values can be efficiently encoded using entropy coding. By doing so, we end up getting compression instead of expansion.

Instead of taking the arithmetic average, we could also form some sort of weighted average. The general procedure would be similar to that described above. (For one of the more well-known weighted average techniques, see [84].)

The representative value does not have to be an average. We could use the pixel values in the approximation at the lower levels of the pyramid as indices into a lookup table. The lookup table can be designed to preserve important information such as edges. The problem with this approach would be the size of the lookup table. If we were using 2×2 blocks of 8-bit values, the lookup table would have 2^{32} values, which is too large for most applications. The size of the table could be reduced if the number of bits per pixel was lower or if, instead of taking 2×2 blocks, we used rectangular blocks of size 2×1 and 1×2 [85].

Finally, we do not have to build the pyramid one layer at a time. After sending the lowest-resolution approximations, we can use some measure of information contained in a block to decide whether it should be transmitted [86]. One possible measure could be the difference between the largest and smallest intensity values in the block. Another might be to look at the maximum number of similar pixels in a block. Using an information measure to guide the progressive transmission of images allows the user to see portions of the image first that are visually more significant.

7.6 Facsimile Encoding

One of the earliest applications of lossless compression in the modern era has been the compression of facsimile, or fax. In facsimile transmission, a page is scanned and converted into a sequence of black or white pixels. The requirements of how fast the facsimile of an A4 document (210×297 mm) must be transmitted have changed over the last two decades. The CCITT (now ITU-T) has issued a number of recommendations based on the speed requirements at a given time. The CCITT classifies the apparatus for facsimile transmission into four groups. Although several considerations are used in this classification, if we only consider the time to transmit an A4-size document over phone lines, the four groups can be described as follows:

- **Group 1:** This apparatus is capable of transmitting an A4-size document in about six minutes over phone lines using an analog scheme. The apparatus is standardized in recommendation T.2.
- **Group 2:** This apparatus is capable of transmitting an A4-size document over phone lines in about three minutes. A Group 2 apparatus also uses an analog scheme and,

therefore, does not use data compression. The apparatus is standardized in recommendation T.3.

- **Group 3:** This apparatus uses a digitized binary representation of the facsimile. Because it is a digital scheme, it can and does use data compression and is capable of transmitting an A4-size document in about a minute. The apparatus is standardized in recommendation T.4.
- **Group 4:** This apparatus has the same speed requirement as Group 3. The apparatus is standardized in recommendations T.6, T.503, T.521, and T.563.

With the arrival of the Internet, facsimile transmission has changed as well. Given the wide range of rates and "apparatus" used for digital communication, it makes sense to focus more on protocols than on apparatus. The newer recommendations from the ITU provide standards for compression that are more or less independent of apparatus.

Later in this chapter, we will look at the compression schemes described in the ITU-T recommendations T.4, T.6, T.82 (JBIG) T.88 (JBIG2), and T.42 (MRC). We begin with a look at an earlier technique for facsimile called *run-length coding*, which still survives as part of the T.4 recommendation.

7.6.1 Run-Length Coding

The model that gives rise to run-length coding is the Capon model [87], a two-state Markov model with states S_w and S_b (S_w corresponds to the case where the pixel that has just been encoded is a white pixel, and S_b corresponds to the case where the pixel that has just been encoded is a black pixel). The transition probabilities $P(w|b)$ and $P(b|w)$, and the probability of being in each state $P(S_w)$ and $P(S_b)$, completely specify this model. For facsimile images, $P(w|w)$ and $P(w|b)$ are generally significantly higher than $P(b|w)$ and $P(b|b)$. The Markov model is represented by the state diagram shown in Figure 7.6.

The entropy of a finite state process with states S_i is given by Equation (2.16). Recall that in Example 2.3.1, the entropy using a probability model and the *iid* assumption was significantly more than the entropy using the Markov model.

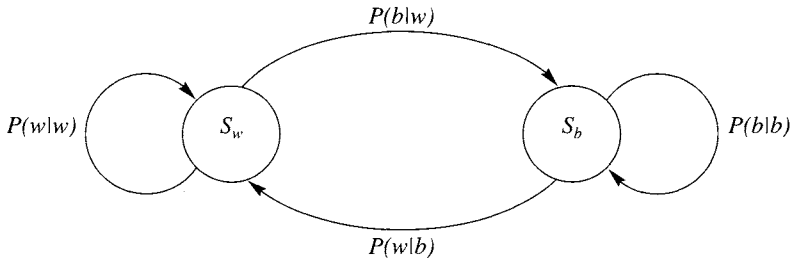


FIGURE 7.6 The Capon model for binary images.

Let us try to interpret what the model says about the structure of the data. The highly skewed nature of the probabilities $P(b|w)$ and $P(w|w)$, and to a lesser extent $P(w|b)$ and $P(b|b)$, says that once a pixel takes on a particular color (black or white), it is highly likely that the following pixels will also be of the same color. So, rather than code the color of each pixel separately, we can simply code the length of the runs of each color. For example, if we had 190 white pixels followed by 30 black pixels, followed by another 210 white pixels, instead of coding the 430 pixels individually, we would code the sequence 190, 30, 210, along with an indication of the color of the first string of pixels. Coding the lengths of runs instead of coding individual values is called run-length coding.

7.6.2 CCITT Group 3 and 4—Recommendations T.4 and T.6

The recommendations for Group 3 facsimile include two coding schemes. One is a one-dimensional scheme in which the coding on each line is performed independently of any other line. The other is two-dimensional; the coding of one line is performed using the line-to-line correlations.

The one-dimensional coding scheme is a run-length coding scheme in which each line is represented as a series of alternating white runs and black runs. The first run is always a white run. If the first pixel is a black pixel, then we assume that we have a white run of length zero.

Runs of different lengths occur with different probabilities; therefore, they are coded using a variable-length code. The approach taken in the CCITT standards T.4 and T.6 is to use a Huffman code to encode the run lengths. However, the number of possible lengths of runs is extremely large, and it is simply not feasible to build a codebook that large. Therefore, instead of generating a Huffman code for each run length r_l, the run length is expressed in the form

$$r_l = 64 \times m + t \qquad \text{for } t = 0, 1, \ldots, 63, \text{ and } m = 1, 2, \ldots, 27. \tag{7.10}$$

When we have to represent a run length r_l, instead of finding a code for r_l, we use the corresponding codes for m and t. The codes for t are called the *terminating codes*, and the codes for m are called the *make-up codes*. If $r_l < 63$, we only need to use a terminating code. Otherwise, both a make-up code and a terminating code are used. For the range of m and t given here, we can represent lengths of 1728, which is the number of pixels per line in an A4-size document. However, if the document is wider, the recommendations provide for those with an optional set of 13 codes. Except for the optional codes, there are separate codes for black and white run lengths. This coding scheme is generally referred to as a *modified Huffman (MH)* scheme.

In the two-dimensional scheme, instead of reporting the run lengths, which in terms of our Markov model is the length of time we remain in one state, we report the transition times when we move from one state to another state. Look at Figure 7.7. We can encode this in two ways. We can say that the first row consists of a sequence of runs 0, 2, 3, 3, 8, and the second row consists of runs of length 0, 1, 8, 3, 4 (notice the first runs of length zero). Or, we can encode the location of the pixel values that occur at a transition from white to

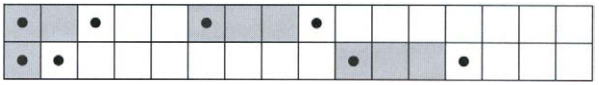


FIGURE 7.7 **Two rows of an image. The transition pixels are marked with a dot.**

black or black to white. The first pixel is an imaginary white pixel assumed to be to the left of the first actual pixel. Therefore, if we were to code transition locations, we would encode the first row as 1, 3, 6, 9 and the second row as 1, 2, 10, 13.

Generally, rows of a facsimile image are heavily correlated. Therefore, it would be easier to code the transition points with reference to the previous line than to code each one in terms of its absolute location, or even its distance from the previous transition point. This is the basic idea behind the recommended two-dimensional coding scheme. This scheme is a modification of a two-dimensional coding scheme called the *Relative Element Address Designate* (READ) code [88, 89] and is often referred to as *Modified READ* (MR). The READ code was the Japanese proposal to the CCITT for the Group 3 standard.

To understand the two-dimensional coding scheme, we need some definitions.

a_0: This is the last pixel whose value is known to both encoder and decoder. At the beginning of encoding each line, a_0 refers to an imaginary white pixel to the left of the first actual pixel. While it is often a transition pixel, it does not have to be.

a_1: This is the first transition pixel to the right of a_0. By definition its color should be the opposite of a_0. The location of this pixel is known only to the encoder.

a_2: This is the second transition pixel to the right of a_0. Its color should be the opposite of a_1, which means it has the same color as a_0. The location of this pixel is also known only to the encoder.

b_1: This is the first transition pixel on the line above the line currently being encoded to the right of a_0 whose color is the opposite of a_0. As the line above is known to both encoder and decoder, as is the value of a_0, the location of b_1 is also known to both encoder and decoder.

b_2: This is the first transition pixel to the right of b_1 in the line above the line currently being encoded.

For the pixels in Figure 7.7, if the second row is the one being currently encoded, and if we have encoded the pixels up to the second pixel, the assignment of the different pixels is shown in Figure 7.8. The pixel assignments for a slightly different arrangement of black and white pixels are shown in Figure 7.9.

If b_1 and b_2 lie between a_0 and a_1, we call the coding mode used the *pass mode*. The transmitter informs the receiver about the situation by sending the code 0001. Upon receipt of this code, the receiver knows that from the location of a_0 to the pixel right below b_2, all pixels are of the same color. If this had not been true, we would have encountered a transition pixel. As the first transition pixel to the right of a_0 is a_1, and as b_2 occurs before a_1, no transitions have occurred and all pixels from a_0 to right below b_2 are the same color. At this time, the last pixel known to both the transmitter and receiver is the pixel below b_2.

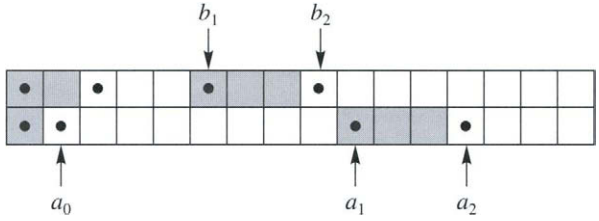


FIGURE 7.8 **Two rows of an image. The transition pixels are marked with a dot.**

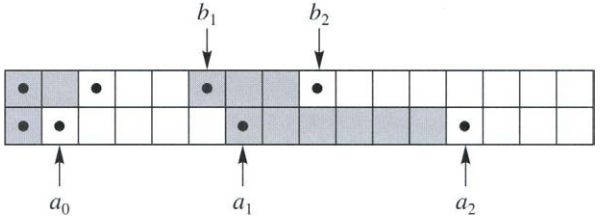


FIGURE 7.9 **Two rows of an image. The transition pixels are marked with a dot.**

Therefore, this now becomes the new a_0, and we find the new positions of b_1 and b_2 by examining the row above the one being encoded and continue with the encoding process.

If a_1 is detected before b_2 by the encoder, we do one of two things. If the distance between a_1 and b_1 (the number of pixels from a_1 to right under b_1) is less than or equal to three, then we send the location of a_1 with respect to b_1, move a_0 to a_1, and continue with the coding process. This coding mode is called the *vertical mode*. If the distance between a_1 and b_1 is large, we essentially revert to the one-dimensional technique and send the distances between a_0 and a_1, and a_1 and a_2, using the modified Huffman code. Let us look at exactly how this is accomplished.

In the vertical mode, if the distance between a_1 and b_1 is zero (that is, a_1 is exactly under b_1), we send the code 1. If the a_1 is to the right of b_1 by one pixel (as in Figure 7.9), we send the code 011. If a_1 is to the right of b_1 by two or three pixels, we send the codes 000011 or 0000011, respectively. If a_1 is to the left of b_1 by one, two, or three pixels, we send the codes 010, 000010, or 0000010, respectively.

In the horizontal mode, we first send the code 001 to inform the receiver about the mode, and then send the modified Huffman codewords corresponding to the run length from a_0 to a_1, and a_1 to a_2.

As the encoding of a line in the two-dimensional algorithm is based on the previous line, an error in one line could conceivably propagate to all other lines in the transmission. To prevent this from happening, the T.4 recommendations contain the requirement that after each line is coded with the one-dimensional algorithm, at most $K - 1$ lines will be coded using the two-dimensional algorithm. For standard vertical resolution, $K = 2$, and for high resolution, $K = 4$.

The Group 4 encoding algorithm, as standardized in CCITT recommendation T.6, is identical to the two-dimensional encoding algorithm in recommendation T.4. The main difference between T.6 and T.4 from the compression point of view is that T.6 does not have a one-dimensional coding algorithm, which means that the restriction described in the previous paragraph is also not present. This slight modification of the modified READ algorithm has earned the name *modified modified READ* (MMR)!

7.6.3 JBIG

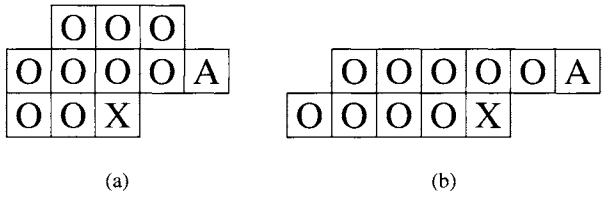
Many bi-level images have a lot of local structure. Consider a digitized page of text. In large portions of the image we will encounter white pixels with a probability approaching 1. In other parts of the image there will be a high probability of encountering a black pixel. We can make a reasonable guess of the situation for a particular pixel by looking at values of the pixels in the neighborhood of the pixel being encoded. For example, if the pixels in the neighborhood of the pixel being encoded are mostly white, then there is a high probability that the pixel to be encoded is also white. On the other hand, if most of the pixels in the neighborhood are black, there is a high probability that the pixel being encoded is also black. Each case gives us a skewed probability—a situation ideally suited for arithmetic coding. If we treat each case separately, using a different arithmetic coder for each of the two situations, we should be able to obtain improvement over the case where we use the same arithmetic coder for all pixels. Consider the following example.

Suppose the probability of encountering a black pixel is 0.2 and the probability of encountering a white pixel is 0.8. The entropy for this source is given by

$$H = -0.2\log_2 0.2 - 0.8\log_2 0.8 = 0.722. \tag{7.11}$$

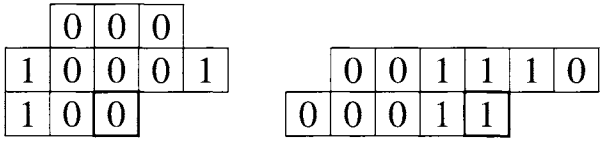
If we use a single arithmetic coder to encode this source, we will get an average bit rate close to 0.722 bits per pixel. Now suppose, based on the neighborhood of the pixels, that we can divide the pixels into two sets, one comprising 80% of the pixels and the other 20%. In the first set, the probability of encountering a white pixel is 0.95, and in the second set the probability of encountering a black pixel is 0.7. The entropy of these sets is 0.286 and 0.881, respectively. If we used two different arithmetic coders for the two sets with frequency tables matched to the probabilities, we would get rates close to 0.286 bits per pixel about 80% of the time and close to 0.881 bits per pixel about 20% of the time. The average rate would be about 0.405 bits per pixel, which is almost half the rate required if we used a single arithmetic coder. If we use only those pixels in the neighborhood that had already been transmitted to the receiver to make our decision about which arithmetic coder to use, the decoder can keep track of which encoder was used to encode a particular pixel.

As we have mentioned before, the arithmetic coding approach is particularly amenable to the use of multiple coders. All coders use the same computational machinery, with each coder using a different set of probabilities. The JBIG algorithm makes full use of this feature of arithmetic coding. Instead of checking to see if most of the pixels in the neighborhood are white or black, the JBIG encoder uses the pattern of pixels in the neighborhood, or *context*, to decide which set of probabilities to use in encoding a particular pixel. If the neighborhood consists of 10 pixels, with each pixel capable of taking on two different values, the number of



(a) (b)

FIGURE 7.10 **(a) Three-line and (b) two-line neighborhoods.**



(a) (b)

FIGURE 7.11 **(a) Three-line and (b) two-line contexts.**

possible patterns is 1024. The JBIG coder uses 1024 to 4096 coders, depending on whether a low- or high-resolution layer is being encoded.

For the low-resolution layer, the JBIG encoder uses one of the two different neighborhoods shown in Figure 7.10. The pixel to be coded is marked **X**, while the pixels to be used for templates are marked **O** or **A**. The **A** and **O** pixels are previously encoded pixels and are available to both encoder and decoder. The **A** pixel can be thought of as a floating member of the neighborhood. Its placement is dependent on the input being encoded. Suppose the image has vertical lines 30 pixels apart. The **A** pixel would be placed 30 pixels to the left of the pixel being encoded. The **A** pixel can be moved around to capture any structure that might exist in the image. This is especially useful in halftone images in which the **A** pixels are used to capture the periodic structure. The location and movement of the **A** pixel are transmitted to the decoder as side information.

In Figure 7.11, the symbols in the neighborhoods have been replaced by 0s and 1s. We take 0 to correspond to white pixels, while 1 corresponds to black pixels. The pixel to be encoded is enclosed by the heavy box. The pattern of 0s and 1s is interpreted as a binary number, which is used as an index to the set of probabilities. The context in the case of the three-line neighborhood (reading left to right, top to bottom) is 0001000110, which corresponds to an index of 70. For the two-line neighborhood, the context is 0011100001, or 225. Since there are 10 bits in these templates, we will have 1024 different arithmetic coders.

In the JBIG standard, the 1024 arithmetic coders are a variation of the arithmetic coder known as the QM coder. The QM coder is a modification of an adaptive binary arithmetic coder called the Q coder [51, 52, 53], which in turn is an extension of another binary adaptive arithmetic coder called the skew coder [90].

In our description of arithmetic coding, we updated the tag interval by updating the endpoints of the interval, $u^{(n)}$ and $l^{(n)}$. We could just as well have kept track of one endpoint

and the size of the interval. This is the approach adopted in the QM coder, which tracks the lower end of the tag interval $l^{(n)}$ and the size of the interval $A^{(n)}$, where

$$A^{(n)} = u^{(n)} - l^{(n)}. \tag{7.12}$$

The tag for a sequence is the binary representation of $l^{(n)}$.

We can obtain the update equation for $A^{(n)}$ by subtracting Equation (4.9) from Equation (4.10) and making this substitution

$$A^{(n)} = A^{(n-1)}(F_X(x_n) - F_X(x_n - 1)) \tag{7.13}$$

$$= A^{(n-1)}P(x_n). \tag{7.14}$$

Substituting $A^{(n)}$ for $u^{(n)} - l^{(n)}$ in Equation (4.9), we get the update equation for $l^{(n)}$:

$$l^{(n)} = l^{(n-1)} + A^{(n-1)}F_X(x_n - 1). \tag{7.15}$$

Instead of dealing directly with the 0s and 1s put out by the source, the QM coder maps them into a More Probable Symbol (MPS) and Less Probable Symbol (LPS). If 0 represents black pixels and 1 represents white pixels, then in a mostly black image 0 will be the MPS, whereas in an image with mostly white regions 1 will be the MPS. Denoting the probability of occurrence of the LPS for the context C by q_c and mapping the MPS to the lower subinterval, the occurrence of an MPS symbol results in the following update equations:

$$l^{(n)} = l^{(n-1)} \tag{7.16}$$

$$A^{(n)} = A^{(n-1)}(1 - q_c) \tag{7.17}$$

while the occurrence of an LPS symbol results in the following update equations:

$$l^{(n)} = l^{(n-1)} + A^{(n-1)}(1 - q_c) \tag{7.18}$$

$$A^{(n)} = A^{(n-1)}q_c. \tag{7.19}$$

Until this point, the QM coder looks very much like the arithmetic coder described earlier in this chapter. To make the implementation simpler, the JBIG committee recommended several deviations from the standard arithmetic coding algorithm. The update equations involve multiplications, which are expensive in both hardware and software. In the QM coder, the multiplications are avoided by assuming that $A^{(n)}$ has a value close to 1, and multiplication with $A^{(n)}$ can be approximated by multiplication with 1. Therefore, the update equations become

For MPS:

$$l^{(n)} = l^{(n-1)} \tag{7.20}$$

$$A^{(n)} = 1 - q_c \tag{7.21}$$

For LPS:

$$l^{(n)} = l^{(n-1)} + (1 - q_c) \tag{7.22}$$

$$A^{(n)} = q_c \tag{7.23}$$

In order not to violate the assumption on $A^{(n)}$ whenever the value of $A^{(n)}$ drops below 0.75, the QM coder goes through a series of rescalings until the value of $A^{(n)}$ is greater than or equal to 0.75. The rescalings take the form of repeated doubling, which corresponds to a left shift in the binary representation of $A^{(n)}$. To keep all parameters in sync, the same scaling is also applied to $l^{(n)}$. The bits shifted out of the buffer containing the value of $l^{(n)}$ make up the encoder output. Looking at the update equations for the QM coder, we can see that a rescaling will occur every time an LPS occurs. Occurrence of an MPS may or may not result in a rescale, depending on the value of $A^{(n)}$.

The probability q_c of the LPS for context C is updated each time a rescaling takes place and the context C is active. An ordered list of values for q_c is listed in a table. Every time a rescaling occurs, the value of q_c is changed to the next lower or next higher value in the table, depending on whether the rescaling was caused by the occurrence of an LPS or an MPS.

In a nonstationary situation, the symbol assigned to LPS may actually occurs more often than the symbol assigned to MPS. This condition is detected when $q_c > (A^{(n)} - q_c)$. In this situation, the assignments are reversed; the symbol assigned the LPS label is assigned the MPS label and vice versa. The test is conducted every time a rescaling takes place.

The decoder for the QM coder operates in much the same way as the decoder described in this chapter, mimicking the encoder operation.

Progressive Transmission

In some applications we may not always need to view an image at full resolution. For example, if we are looking at the layout of a page, we may not need to know what each word or letter on the page is. The JBIG standard allows for the generation of progressively lower-resolution images. If the user is interested in some gross patterns in the image (for example, if they were interested in seeing if there were any figures on a particular page) they could request a lower-resolution image, which could be transmitted using fewer bits. Once the lower-resolution image was available, the user could decide whether a higher-resolution image was necessary. The JBIG specification recommends generating one lower-resolution pixel for each 2×2 block in the higher-resolution image. The number of lower-resolution images (called layers) is not specified by JBIG.

A straightforward method for generating lower-resolution images is to replace every 2×2 block of pixels with the average value of the four pixels, thus reducing the resolution by two in both the horizontal and vertical directions. This approach works well as long as three of the four pixels are either black or white. However, when we have two pixels of each kind, we run into trouble; consistently replacing the four pixels with either a white or black pixel causes a severe loss of detail, and randomly replacing with a black or white pixel introduces a considerable amount of noise into the image [81].

Instead of simply taking the average of every 2×2 block, the JBIG specification provides a table-based method for resolution reduction. The table is indexed by the neighboring pixels shown in Figure 7.12, in which the circles represent the lower-resolution layer pixels and the squares represent the higher-resolution layer pixels.

Each pixel contributes a bit to the index. The table is formed by computing the expression

$$4e + 2(b + d + f + h) + (a + c + g + i) - 3(B + C) - A.$$

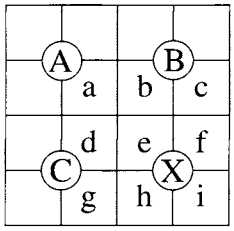


FIGURE 7. 12 Pixels used to determine the value of a lower-level pixel.

If the value of this expression is greater than 4.5, the pixel X is tentatively declared to be 1. The table has certain exceptions to this rule to reduce the amount of edge smearing, generally encountered in a filtering operation. There are also exceptions that preserve periodic patterns and dither patterns.

As the lower-resolution layers are obtained from the higher-resolution images, we can use them when encoding the higher-resolution images. The JBIG specification makes use of the lower-resolution images when encoding the higher-resolution images by using the pixels of the lower-resolution images as part of the context for encoding the higher-resolution images. The contexts used for coding the lowest-resolution layer are those shown in Figure 7.10. The contexts used in coding the higher-resolution layer are shown in Figure 7.13.

Ten pixels are used in each context. If we include the 2 bits required to indicate which context template is being used, 12 bits will be used to indicate the context. This means that we can have 4096 different contexts.

Comparison of MH, MR, MMR, and JBIG

In this section we have seen three old facsimile coding algorithms: modified Huffman, modified READ, and modified modified READ. Before we proceed to the more modern techniques found in T.88 and T.42, we compare the performance of these algorithms with the earliest of the modern techniques, namely JBIG. We described the JBIG algorithm as an application of arithmetic coding in Chapter 4. This algorithm has been standardized in ITU-T recommendation T.82. As we might expect, the JBIG algorithm performs better than the MMR algorithm, which performs better than the MR algorithm, which in turn performs better than the MH algorithm. The level of complexity also follows the same trend, although we could argue that MMR is actually less complex than MR.

A comparison of the schemes for some facsimile sources is shown in Table 7.4. The modified READ algorithm was used with $K = 4$, while the JBIG algorithm was used with an adaptive three-line template and adaptive arithmetic coder to obtain the results in this table. As we go from the one-dimensional MH coder to the two-dimensional MMR coder, we get a factor of two reduction in file size for the sparse text sources. We get even more reduction when we use an adaptive coder and an adaptive model, as is true for the JBIG coder. When we come to the dense text, the advantage of the two-dimensional MMR over the one-dimensional MH is not as significant, as the amount of two-dimensional correlation becomes substantially less.

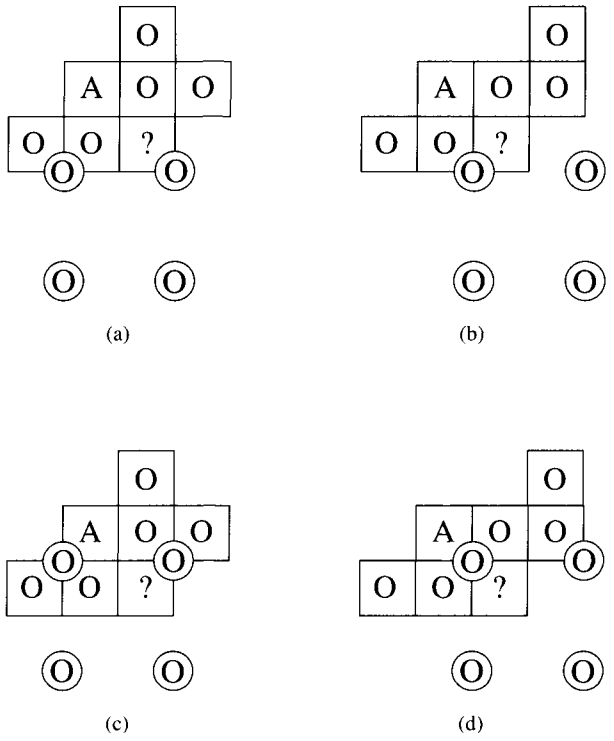


FIGURE 7.13 Contexts used in the coding of higher-resolution layers.

TABLE 7.4 Comparison of binary image coding schemes. Data from [91].

Source Description	Original Size (pixels)	MH (bytes)	MR (bytes)	MMR (bytes)	JBIG (bytes)
Letter	4352 × 3072	20,605	14,290	8,531	6,682
Sparse text	4352 × 3072	26,155	16,676	9,956	7,696
Dense text	4352 × 3072	135,705	105,684	92,100	70,703

The compression schemes specified in T.4 and T.6 break down when we try to use them to encode halftone images. In halftone images, gray levels are represented using binary pixel patterns. A gray level closer to black would be represented by a pattern that contains more black pixels, while a gray level closer to white would be represented by a pattern with fewer black pixels. Thus, the model that was used to develop the compression schemes specified in T.4 and T.6 is not valid for halftone images. The JBIG algorithm, with its adaptive model and coder, suffers from no such drawbacks and performs well for halftone images also [91].

7.6.4 JBIG2—T.88

The JBIG2 standard was approved in February of 2000. Besides facsimile transmission, the standard is also intended for document storage, archiving, wireless transmission, print spooling, and coding of images on the Web. The standard provides specifications only for the decoder, leaving the encoder design open. This means that the encoder design can be constantly refined, subject only to compatibility with the decoder specifications. This situation also allows for lossy compression, beacuse the encoder can incorporate lossy transformations to the data that enhance the level of compression.

The compression algorithm in JBIG provides excellent compression of a generic bi-level image. The compression algorithm proposed for JBIG2 uses the same arithmetic coding scheme as JBIG. However, it takes advantage of the fact that a significant number of bi-level images contain structure that can be used to enhance the compression performance. A large percentage of bi-level images consist of text on some background, while another significant percentage of bi-level images are or contain halftone images. The JBIG2 approach allows the encoder to select the compression technique that would provide the best performance for the type of data. To do so, the encoder divides the page to be compressed into three types of regions called *symbol regions*, *halftone regions*, and *generic regions*. The symbol regions are those containing text data, the halftone regions are those containing halftone images, and the generic regions are all the regions that do not fit into either category.

The partitioning information has to be supplied to the decoder. The decoder requires that all information provided to it be organized in *segments* that are made up of a segment header, a data header, and segment data. The page information segment contains information about the page including the size and resolution. The decoder uses this information to set up the page buffer. It then decodes the various regions using the appropriate decoding procedure and places the different regions in the appropriate location.

Generic Decoding Procedures

There are two procedures used for decoding the generic regions: the generic region decoding procedure and the generic refinement region decoding procedure. The generic region decoding procedure uses either the MMR technique used in the Group 3 and Group 4 fax standards or a variation of the technique used to encode the lowest-resolution layer in the JBIG recommendation. We describe the operation of the MMR algorithm in Chapter 6. The latter procedure is described as follows.

The second generic region decoding procedure is a procedure called *typical prediction*. In a bi-level image, a line of pixels is often identical to the line above. In typical prediction, if the current line is the same as the line above, a bit flag called $LNTP_n$ is set to 0, and the line is not transmitted. If the line is not the same, the flag is set to 1, and the line is coded using the contexts currently used for the low-resolution layer in JBIG. The value of $LNTP_n$ is encoded by generating another bit, $SLNTP_n$, according to the rule

$$SLNTP_n = !(LNTP_n \oplus LNTP_{n-1})$$

which is treated as a virtual pixel to the left of each row. If the decoder decodes an *LNTP* value of 0, it copies the line above. If it decodes an *LNTP* value of 1, the following bits

in the segment data are decoded using an arithmetic decoder and the contexts described previously.

The generic refinement decoding procedure assumes the existence of a *reference* layer and decodes the segment data with reference to this layer. The standard leaves open the specification of the reference layer.

Symbol Region Decoding

The symbol region decoding procedure is a dictionary-based decoding procedure. The symbol region segment is decoded with the help of a symbol dictionary contained in the symbol dictionary segment. The data in the symbol region segment contains the location where a symbol is to be placed, as well as the index to an entry in the symbol dictionary. The symbol dictionary consists of a set of bitmaps and is decoded using the generic decoding procedures. Note that because JBIG2 allows for lossy compression, the symbols do not have to exactly match the symbols in the original document. This feature can significantly increase the compression performance when the original document contains noise that may preclude exact matches with the symbols in the dictionary.

Halftone Region Decoding

The halftone region decoding procedure is also a dictionary-based decoding procedure. The halftone region segment is decoded with the help of a halftone dictionary contained in the halftone dictionary segment. The halftone dictionary segment is decoded using the generic decoding procedures. The data in the halftone region segment consists of the location of the halftone region and indices to the halftone dictionary. The dictionary is a set of fixed-size halftone patterns. As in the case of the symbol region, if lossy compression is allowed, the halftone patterns do not have to exactly match the patterns in the original document. By allowing for nonexact matches, the dictionary can be kept small, resulting in higher compression.

7.7 MRC—T.44

With the rapid advance of technology for document production, documents have changed in appearance. Where a document used to be a set of black and white printed pages, now documents contain multicolored text as well as color images. To deal with this new type of document, the ITU-T developed the recommendation T.44 for Mixed Raster Content (MRC). This recommendation takes the approach of separating the document into elements that can be compressed using available techniques. Thus, it is more an approach of partitioning a document image than a compression technique. The compression strategies employed here are borrowed from previous standards such as JPEG (T.81), JBIG (T.82), and even T.6.

The T.44 recommendation divides a page into slices where the width of the slice is equal to the width of the entire page. The height of the slice is variable. In the base mode, each

FIGURE 7. 14 Ruby's birthday invitation.

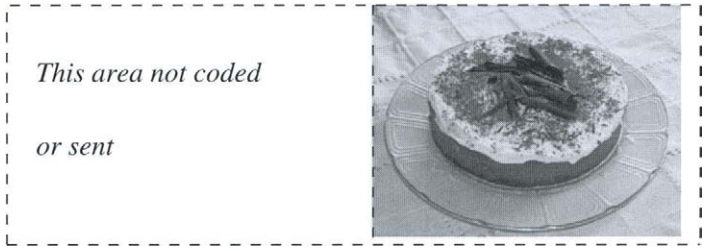


FIGURE 7. 15 The background layer.

slice is represented by three layers: a background layer, a foreground layer, and a mask layer. These layers are used to effectively represent three basic data types: color images (which may be continuous tone or color mapped), bi-level data, and multilevel (multicolor) data. The multilevel image data is put in the background layer, and the mask and foreground layers are used to represent the bi-level and multilevel nonimage data. To work through the various definitions, let us use the document shown in Figure 7.14 as an example. We have divided the document into two slices. The top slice contains the picture of the cake and two lines of writing in two "colors." Notice that the heights of the two slices are not the same and the complexity of the information contained in the two slices is not the same. The top slice contains multicolored text and a continuous tone image whereas the bottom slice contains only bi-level text. Let us take the upper slice first and see how to divide it into the three layers. We will discuss how to code these layers later. The background layer consists of the cake and nothing else. The default color for the background layer is white (though this can be changed). Therefore, we do not need to send the left half of this layer, which contains only white pixels.

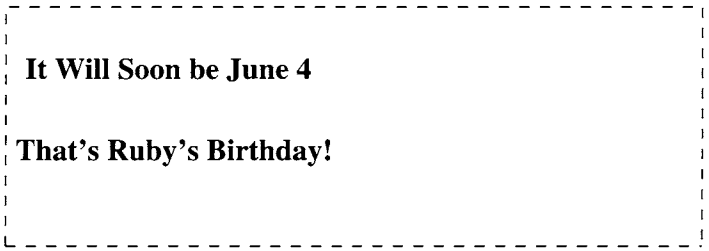


FIGURE 7.16 The mask layer.

FIGURE 7.17 The foreground layer.

The mask layer (Figure 7.16) consists of a bi-level representation of the textual information, while the foreground layer contains the colors used in the text. To reassemble the slice we begin with the background layer. We then add to it pixels from the foreground layer using the mask layer as the guide. Wherever the mask layer pixel is black (1) we pick the corresponding pixel from the foreground layer. Wherever the mask pixel is white (0) we use the pixel from the background layer. Because of its role in selecting pixels, the mask layer is also known as the selector layer. During transmission the mask layer is transmitted first, followed by the background and the foreground layers. During the rendering process the background layer is rendered first.

When we look at the lower slice we notice that it contains only bi-level information. In this case we only need the mask layer because the other two layers would be superfluous. In order to deal with this kind of situation, the standard defines three different kinds of stripes. Three-layer stripes (3LS) contain all three layers and is useful when there is both image and textual data in the strip. Two-layer stripes (2LS) only contain two layers, with the third set to a constant value. This kind of stripe would be useful when encoding a stripe with multicolored text and no images, or a stripe with images and bi-level text or line drawings. The third kind of stripe is a one-layer stripe (1LS) which would be used when a stripe contains only bi-level text or line art, or only continuous tone images.

Once the document has been partitioned it can be compressed. Notice that the types of data we have after partitioning are continuous tone images, bi-level information, and multilevel regions. We already have efficient standards for compressing these types of data. For the mask layer containing bi-level information, the recommendation suggests that one of several approaches can be used, including modified Huffman or modified READ

(as described in recomendation T.4), MMR (as described in recommendation T.6) or JBIG (recommendation T.82). The encoder includes information in the datastream about which algorithm has been used. For the continuous tone images and the multilevel regions contained in the foreground and background layers, the recommendation suggests the use of the JPEG standard (recommendation T.81) or the JBIG standard. The header for each slice contains information about which algorithm is used for compression.

7.8 Summary

In this section we have examined a number of ways to compress images. All these approaches exploit the fact that pixels in an image are generally highly correlated with their neighbors. This correlation can be used to predict the actual value of the current pixel. The prediction error can then be encoded and transmitted. Where the correlation is especially high, as in the case of bi-level images, long stretches of pixels can be encoded together using their similarity with previous rows. Finally, by identifying different components of an image that have common characteristics, an image can be partitioned and each partition encoded using the algorithm best suited to it.

Further Reading

1. A detailed survey of lossless image compression techniques can be found in "Lossless Image Compression" by K.P. Subbalakshmi. This chapter appears in the *Lossless Compression Handbook, Academic Press*, 2003.
2. For a detailed description of the LOCO-I and JPEG-LS compression algorithm, see "The LOCO-I Lossless Image Compression Algorithm: Principles and Standardization into JPEG-LS," Hewlett-Packard Laboratories Technical Report HPL-98-193, November 1998 [92].
3. The JBIG and JBIG2 standards are described in a very accessible manner in "Lossless Bilevel Image Compression," by M.W. Hoffman. This chapter appears in the *Lossless Compression Handbook, Academic Press*, 2003.
4. The area of lossless image compression is a very active one, and new schemes are being published all the time. These articles appear in a number of journals, including *Journal of Electronic Imaging*, *Optical Engineering*, *IEEE Transactions on Image Processing*, *IEEE Transactions on Communications*, *Communications of the ACM*, *IEEE Transactions on Computers*, and *Image Communication*, among others.

7.9 Projects and Problems

1. Encode the binary image shown in Figure 7.18 using the modified Huffman scheme.
2. Encode the binary image shown in Figure 7.18 using the modified READ scheme.
3. Encode the binary image shown in Figure 7.18 using the modified modified READ scheme.

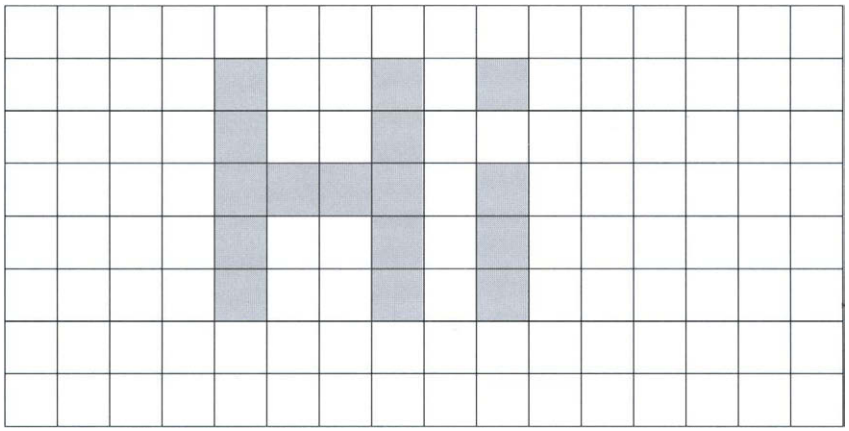


FIGURE 7.18 **An 8 x 16 binary image.**

4. Suppose we want to transmit a 512×512, 8-bits-per-pixel image over a 9600 bits per second line.

   (a) If we were to transmit this image using raster scan order, after 15 seconds how many rows of the image will the user have received? To what fraction of the image does this correspond?

   (b) If we were to transmit the image using the method of Example 7.5.1, how long would it take the user to receive the first approximation? How long would it take to receive the first two approximations?

5. An implementation of the progressive transmission example (Example 7.5.1) is included in the programs accompanying this book. The program is called `prog_tran1.c`. Using this program as a template, experiment with different ways of generating approximations (you could use various types of weighted averages) and comment on the qualitative differences (or lack thereof) with using various schemes. Try different block sizes and comment on the practical effects in terms of quality and rate.

6. The program `jpegll_enc.c` generates the residual image for the different JPEG prediction modes, while the program `jpegll_dec.c` reconstructs the original image from the residual image. The output of the encoder program can be used as the input to the public domain arithmetic coding program mentioned in Chapter 4 and the Huffman coding programs mentioned in Chapter 3. Study the performance of different combinations of prediction mode and entropy coder using three images of your choice. Account for any differences you see.

7. Extend `jpegll_enc.c` and `jpegll_dec.c` with an additional prediction mode—be creative! Compare the performance of your predictor with the JPEG predictors.

8. Implement the portions of the CALIC algorithm described in this chapter. Encode the Sena image using your implementation.