

Chapter 3:

Methodology

3. Methodology

This chapter surveys and explains how the implemented compression methods work and addresses issues related to the implementation. The concept of arithmetic coding is introduced in Section 3.1. A practical implementation of arithmetic coding is described in Section 3.2. An implementation that improves upon this, and which is used in this dissertation, is described in Section 3.3. Arithmetic coding requires the use of modeling, as explained in Section 3.4. Section 3.5 details the use of context-based statistics modeling. Two methods involving context modeling which are used in this dissertation, bitplane coding and PPM, are described in Section 3.6 and 3.7 respectively. A method combining bitplane coding and PPM is outlined in Section 3.8. Section 3.9 explains how the images that are used for testing the implementations are obtained. Finally, the implementation environment is described in Section 3.10.

3.1 Arithmetic coding concept

In arithmetic coding, a *tag* that uniquely represents the entire sequence to be encoded is generated. This is in contrast with assigning codes to the individual symbols, for example in Huffman coding. The tag distinguishes the sequence from all other possible sequences, so that the sequence can be decoded. The following description was adapted from Sayood (2000) and Witten *et al.* (1987).

3.1.1 Tag generation process

To show conceptually how it works, the interval of real numbers $[0,1)$ can be used as the set which contains the possible tags. $[0,1)$ denotes all values of x with $0 \leq x < 1$. Consider a sequence of symbols comprising the source data or *message* to be

compressed. Associated with each symbol is a probability of its occurrence (how the probability is found would depend on the modeling portion).

The procedure of finding the tag works as follows:

1. Start with a current interval $[L,H]$ initialized to the value $[0,1]$. L and H are variables representing the low and high limits of the interval.
2. Repeat, for each symbol in the message sequence:
 - i. subdivide the current interval into subintervals, one for representing each possible symbol. The size of the subinterval is proportional to the probability of the symbol
 - ii. the subinterval for the symbol that occurred is selected to be the new current interval.
3. When the entire sequence has been processed, the output should be a number that uniquely identifies the current interval.

Any number in the final interval can be used as the tag, since the appearance of each new symbol restricts the tag to a subinterval that is disjoint from any other interval that may have been generated.

Example: Consider the alphabet $S = \{s_1, s_2, s_3\}$ and the sequence to be encoded is $s_1 s_1 s_3$. Their probabilities are given as $P(s_1) = 0.7$, $P(s_2) = 0.2$ and $P(s_3) = 0.1$. The interval is divided among the three symbols by assigning a subinterval proportional in size to its probability, as shown in Figure 3.1. At the start, the subintervals corresponding to s_3 , s_2 , and s_1 respectively are $[0,0.1)$, $[0.1,0.3)$ and $[0.3,1.0)$. Encoding the sequence then proceeds as shown in Table 3.1. The final interval is $[0.51, 0.559)$. Any number in this final interval can be taken to represent the sequence $s_1 s_1 s_3$ uniquely, for example 0.55.

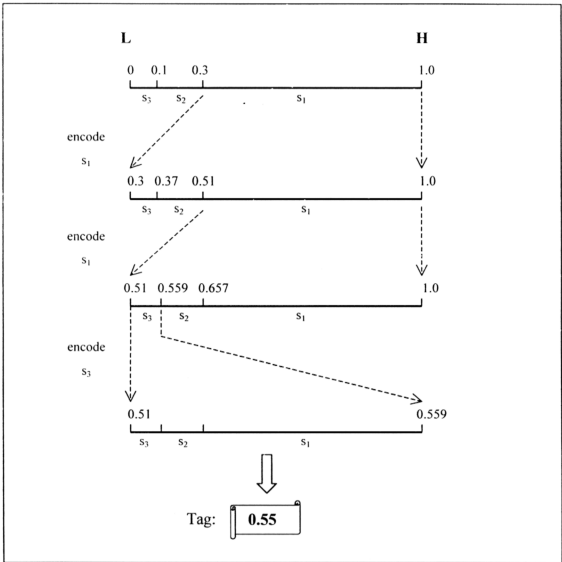


Figure 3.1: Example of subdividing intervals in arithmetic coding

Table 3.1: Subinterval division process

current interval	encode symbol	subintervals		
		s_3	s_2	s_3
[0,1.0)	s_1	[0, 0.1)	[0.1, 0.3)	[0.3, 1.0)
[0.3,1.0)	s_1	[0.3,0.37)	[0.37, 0.51)	[0.51, 1.0)
[0.51,1.0)	s_3	[0.51, 0.559)	[0.559, 0.657)	[0.657, 1.0)
[0.51,0.559)				

In practice, it is sufficient to calculate the lower and upper limits, L and H, of the subinterval containing the symbol to be encoded. When encoding the symbol s_n , L and H can be updated using the following equations:

$$L' = L + (H - L) \times F_s(n) \tag{3.1}$$

$$H' = L + (H - L) \times F_s(n-1) \tag{3.2}$$

where, L' and H' are the new lower and upper limit respectively. $F_s(n)$ is given by:

$$F_s(n) = 1 - \sum_{i=1}^n P(s_i) \qquad \text{for } n > 0 \tag{3.3}$$

$$F_s(n) = 1 \qquad \text{for } n = 0 \tag{3.4}$$

where $P(s_i)$ denotes the probability of symbol s_i occurring.

Example: Table 3.2 shows the probability table storing the values of $P(s_i)$ and $F_s(n)$ for the previous example. Table 3.3 shows how the subintervals are found.

Table 3.2: Probability table

n	s_i	$P(s_i)$	$F_s(n)$
0	-	-	1.0
1	s_1	0.7	0.3
2	s_2	0.2	0.1
3	s_3	0.1	0

Table 3.3: Subinterval limits calculation

current limits		encode symbol	new limits	
L	H		L'	H'
0	1.0	s_1	$= 0 + (1-0) \times 0.3 = 0.3$	$= 0 + (1-0) \times 1 = 1.0$
0.3	1.0	s_1	$= 0.3 + (1-0.3) \times 0.3 = 0.51$	$= 0.3 + (1-0.3) \times 1 = 1.0$
0.51	1.0	s_3	$= 0.51 + (1-0.51) \times 0 = 0.51$	$= 0.51 + (1-0.51) \times 0.1 = 0.559$
0.51	0.559			

3.1.2 **Decoding**

The message can be decoded from the tag using the following steps:

- 1. Initialize the lower and upper limits, $L = 0$ and $H = 1$
- 2. Find the value $target = (tag - L)/(H - L)$
- 3. Find the value of n for which $F_s(n) \leq target < F_s(n+1)$
- 4. The decoded symbol is s_n
- 5. Find the new limits L' and H' using the equations 3.1 and 3.2
- 6. Repeat steps 2-5 until the entire message has been decoded

To know when decoding is complete and should be stopped, an end-of-message symbol can be sent by the encoder to indicate the end of the message. Alternatively, the decoder is informed beforehand the length of the message and stops decoding when enough number of symbols have been decoded.

Example: From the previous example, let the tag representing the sequence be 0.51. The decoding process is shown in Table 3.4.

Table 3.4: Decoding process

current interval		target	new current interval		decoded symbol
L	H		L'	H'	
0	1.0	$= (0.51-0)/(1-0)$ $= 0.51$	$= 0+(1-0) \times 0.3$ $= 0.3$	$= 0 + (1-0) \times 1$ $= 1.0$	s_1
0.3	1.0	$= (0.51-0.3)/(1-0.3)$ $= 0.3$	$= 0.3 + (1-0.3) \times 0.3$ $= 0.51$	$= 0.3 + (1-0.3) \times 1$ $= 1.0$	s_1
0.51	1.0	$= (0.51-0.51)/(1-0.51)$ $= 0$	$= 0.51 + (1-0.51) \times 0$ $= 0.51$	$= 0.51 + (1-0.51) \times 0.1$ $= 0.559$	s_3
0.51	0.559				

3.2 Integer implementation

A practical implementation for a computer has to use integer arithmetic instead of floating-point arithmetic. The following implementation is due to Witten *et al.* (1987).

Let b be the number of bits used to represent an integer number. There are 2^b possible integer numbers, from 0 to 2^b-1 . The initial interval $[0,1)$ can thus be mapped to the interval $[0,2^b)$. For example, if $b=16$, then the range used for integer implementation is $[0,65536)$.

Let $freq(s_i)$ be the number of times the symbol s_i has occurred in a sequence of length $total_count$. $P(s_i)$ can thus be estimated by $freq(s_i)/total_count$. Equation 3.3 then becomes:

$$\begin{aligned} F_s(n) &= 1 - \left[\sum_{i=1}^n freq(s_i) \right] / total_count \\ &= cum_freq(n) / total_count \end{aligned} \quad (3.5)$$

$$\text{where } cum_freq(n) = total_count - \sum_{i=1}^n freq(s_i)$$

For the integer implementation, equation 3.1 and 3.2 becomes:

$$L' = L + \left\lfloor \frac{R \times cum_freq(n)}{total_count} \right\rfloor \quad (3.6)$$

$$H' = L + \left\lfloor \frac{R \times cum_freq(n-1)}{total_count} \right\rfloor + 1 \quad (3.7)$$

where $R = H - L + 1$, and $\lfloor x \rfloor$ is the largest integer less than or equal to x .

3.2.1 Renormalization and incremental transmission

For each symbol processed during encoding, the resulting interval becomes smaller and smaller. It takes more bits to represent the tag for a small interval compared to a larger interval. To implement the encoding process in a computer, the precision required to represent an interval becomes higher as the length of the sequence increases. As a computer will have limited precision (depending on the number of bits allocated to represent a number), the values of L and H are bound to become closer and converge when the interval becomes very small. Information about the sequence will be lost from the point in which the two values converged. To avoid this, the interval needs to be renormalized back to the original range.

First, consider the floating point arithmetic implementation. When interval becomes smaller, there are three possibilities:

- i. the interval is entirely confined to the lower half of the interval, [0,0.5)
- ii. the interval is entirely confined to the upper half of the interval, [0.5,1)
- iii.the interval straddles the midpoint of the interval, [0.25, 0.75)

To renormalize, the interval is remapped into the [0, 1) interval as shown in Table 3.5.

Table 3.5: Renormalization for floating point arithmetic

condition	mapping	equation
If $H < 0.5$	$E_1: [0, 0.5) \rightarrow [0, 1)$	$E_1(x) = 2x$
If $L > 0.5$	$E_2: [0.5, 1) \rightarrow [0, 1)$	$E_2(x) = 2(x-0.5)$
If $H < 0.75$ and $L > 0.25$	$E_3: [0.25, 0.75) \rightarrow [0, 1)$	$E_3(x) = 2(x-0.25)$

For the interval $[0,0.5)$, the most significant bit of the binary representation of all numbers in that interval is 0. For the interval $[0.5,1)$, the most significant bit of the binary representation of all numbers in that interval is 1. Therefore, once the interval is restricted to either the lower or upper half, the most significant bit of the tag is fully determined. It can be shifted out and transmitted to the decoder, thus making incremental transmission of the encoded bitstream possible.

The mapping E3 is required because L and H can become close to each other without triggering the mappings E1 and E2. This happens when L and H moves towards 0.5, but L remains in the lower half while H remains in the upper half.

In the integer implementation, the initial range is $[0, 2^b)$. If b is the number of bits representing the integer number, the binary representation for the initial range is:

$$L = 0 = \underbrace{(000\dots0)}_{b \text{ zeroes}}_2$$

$$H = 2^b - 1 = \underbrace{(111\dots1)}_{b \text{ ones}}_2$$

The mapping for integer implementation is given in Table 3.6. The mapping enables incremental transmission. The E_1 and E_2 mapping operation is equivalent to shifting the integer numbers one bit to the left, thus shifting out the MSB (0 and 1 respectively for E_1 and E_2). The bit shifted into the LSB is 0 and 1 respectively for L and H.

For E3 mapping, the following is done. The integer numbers are also shifted one bit to the left, shifting out the MSB. However, instead of immediately transmitting a 1 or 0, the number of times that E3 mappings are done consecutively is kept track. The first time an E_1 or E_2 mapping is performed after the E3 mappings, the same number of bits

is output, with a value that is complementary to the bit output by E1 or E2. For example, if three E3 mappings are done, followed by an E1 mapping, then the bits output are 0111. If it was an E2 mapping that followed the E3 mappings, then the bits output are 1000.

Table 3.6: Renormalization for integer arithmetic

condition	mapping	equation
If $H < Half$	$E_1: [0, Half) \rightarrow [0, 2^b)$	$E_1(x) = 2x$
If $L > Half$	$E_2: [Half, 2^b) \rightarrow [0, 2^b)$	$E_2(x) = 2(x - Half)$
If $H < Third_qtr$ and $L > First_qtr$	$E_3: [First_qtr, Third_qtr) \rightarrow [0, 2^b)$	$E_3(x) = 2(x - First_qtr)$

*Where:

$$\begin{aligned}
 Half &= 2^b/2 = 2^{b-1} \\
 First_qtr &= 2^b/4 = 2^{b-2} \\
 Third_qtr &= 3 \times 2^b/4 = 3 \times 2^{b-2}
 \end{aligned}$$

3.2.2 Decoding

The corresponding decoder for the integer implementation above is as follows. The sequence can be decoded from the tag using the following steps:

- [L, H) is initialized to $[0, 2^b)$ as in the encoder. A number, V, is used is to receive the bitstream. Bits that have been decoded are shifted out of the left-most end (MSB) and newly received bits are shifted into the right-most end (LSB).
- An integer number, *target*, is found. The number lies in the range $[cum_freq(n), cum_freq(n-1))$ that was used at the corresponding step at the encoder.

$$target = \left\lfloor \frac{(V - L + 1) \times total_count - 1}{R} \right\rfloor \tag{3.8}$$

- From the value of *target*, the symbol corresponding to it is found i.e. the symbol s_n for which $cum_freq(n) \leq target < cum_freq(n-1)$

4. Find the new current interval L' and H' using the equations 3.6 and 3.7.
5. Do renormalization if required, using the mapping described in the previous section, shifting in new bits into V . Repeat renormalization until it is not required.
6. Continue 2-5 until the entire sequence has been decoded.

3.3 Improved implementation

In the implementation above by Witten *et al.* (1987), the state of the coder is recorded by the value L and H , to denote the interval's range $[L, H]$. The interval range $R = H - L + 1$ is calculated at each step. Moffat *et al.* (1998) proposed improvements to the implementation by rearranging the calculation of the interval range. A slightly different way was used to record the coder's interval limits by using L and R , instead of L and H . Thus the interval range is given by $[L, L+R)$. L and R are initially 0 and 2^{b-1} respectively.

To find the new current interval, L' and R' is given by:

$$L' = \left\lfloor \frac{L + R \times \text{cum_freq}(n)}{\text{total_count}} \right\rfloor \quad (3.9)$$

$$R' = \left\lfloor \frac{R \times \text{cum_freq}(n-1)}{\text{total_count}} \right\rfloor - \left\lfloor \frac{R \times \text{cum_freq}(n)}{\text{total_count}} \right\rfloor \quad (3.10)$$

Define

$$T = \left\lfloor \frac{R \times \text{cum_freq}(n)}{\text{total_count}} \right\rfloor \quad (3.11)$$

Then,

$$L' = L + T \quad (3.12)$$

$$R' = \left\lfloor \frac{R \times \text{cum_freq}(n-1)}{\text{total_count}} \right\rfloor - T \quad (3.13)$$

3.3.1 Improvements

Moffat *et al.* (1998) cites the following improvements:

1. The number of multiplications performed was reduced to one in the encoder and two in the decoder. When the value of H is equal to *total_count*, a further operation is saved in both the encoder and decoder if the remainder of the range is allocated to the last symbol in the alphabet. These savings result in faster execution of the arithmetic coder.
2. The rearrangement of the multiplicative operations allowed larger frequency counts to be manipulated. If f and b are the number of bits used respectively to represent the frequency counts and the coder limits L and R , then the constraints of the maximum frequency count is given by $f \leq b-2$, and $total_count \leq 2^f$. For example, if $b=32$, then $f=30$ can be used, allowing larger amount of symbols to be processed before frequency count scaling (the reduction of frequency counts) is required. In contrast, Witten *et al.*'s implementation is constrained by $2f+1 \leq b$.
3. The decoder implementation was reorganized to allow a simpler decoder renormalization loop, resulting in faster execution.
4. Implementation using add and shift operations instead of multiplication and division became possible, thereby possibly increasing execution speed.
5. An extension for arithmetic coding of binary alphabets was also provided.

3.3.2 Restrictions

There are however several restrictions to be aware of when using the improved arithmetic coder (Moffat *et al.*, 1998):

1. The rearrangements in the arithmetic coder resulted in a slight loss of compression effectiveness, but can be minimized by having the symbol with the largest

probability (the most probable symbol) maintained at the top of the probability table. The alternative is to use a large value of $b \cdot f$

2. For better compression, the value of f should be as small as possible. This however would result in more frequent count scaling. Therefore, the tradeoff should be considered.
3. If add and shift operations are used, the frequency counts need to be kept in partially normalized form such that $2^{f-1} < t \leq 2^{f-6}$.
4. Using add and shift operations may not result in much improvement to execution time, depending on the machine architecture (how much faster add and shift is compared to multiplication and division). On a Pentium Pro machine, it was found that the add/shift implementation offered only slight improvement of execution time.

The implementation of the compressors in this dissertation used the improved arithmetic coder by Moffat *et al.* (1998). The implementation allows the choice of either the add/shift or multiplication/division approach. The latter was chosen since restrictions 3.3.2(3) and 3.3.2(4) above complicates the implementation somewhat and compression speed is a secondary issue anyway. The implementation can be modified later if tuning for improved speed performance is required.

Figure 3.2 shows the pseudocode for the arithmetic encoder of Moffat *et al.*'s arithmetic coding implementation. The functions `start_encode()` and `finish_encode()` initialize and terminate the arithmetic encoder respectively. The function `arithmetic_encode()` encodes a symbol given l , h and t , which denote $cum_freq(n)$, $cum_freq(n-1)$, and $total_count$ respectively.

```

function start_encode( )


---


L ← 0
R ←  $2^{b-1}$ 
g ← 0

function arithmetic_encode(l, h, t)


---


r ← R/t
L ← L+r*1
if (h<t)
    R ← r*(h-1)
else
    R ← R-r*1

/*renormalization*/
while (R ≤  $2^{b-2}$ ) {
    if (L+R ≤  $2^{b-1}$ ) {
        output 0 once
        output 1 g times
        g ← 0
    } else {
        if (L ≥  $2^{b-1}$ ) {
            output 1 once
            output 0 g times
            g ← 0
            L ← L- $2^{b-1}$ 
        } else {
            g ← g+1
            L ← L- $2^{b-2}$ 
        }
    }
    L ← 2*L
    R ← 2*R
}

function finish_encode( )


---


bits ← L
for (i = 1 to b) {
    output (bits>>(b-i))AND 12
    output the opposite bit g times
    g ← 0
}

```

Figure 3.2: Pseudocode for arithmetic encoder of Moffat *et al.*

Figure 3.3 shows the pseudocode for the arithmetic decoder of Moffat *et al.*'s arithmetic coding implementation. The functions `start_decode()` and `finish_decode()` initialize and terminate the arithmetic decoder respectively. The function `decode_target()` returns the target value, while `arithmetic_decode()` updates the decoder's state.

The source code for the arithmetic coder was made readily available by the authors (at http://www.cs.mu.oz.au/~alistair/arith_coder/). The functions related to the arithmetic coder were used in this dissertation.

```

function start_decode(l, h, t)


---


R ← 2b-1
D ← 0
for (i=1 to b)
    D ← 2*D + next_bit

function decode_target(t)


---


r ← R/t
target ← minimum{t-1, D/r}
return target

function arithmetic_decode(l, h, t)


---


D ← D-r*1
if (h<t)
    R ← r*(h-1)
else
    R ← R-r*1

/*renormalization*/
while (R ≤ 2b-2) {
    R ← 2*R
    D ← 2*D + next_bit
}

function finish_decode()


---


/* does nothing */

```

Figure 3.3: Pseudocode for arithmetic decoder of Moffat *et al.*

3.3.3 Binary arithmetic coder

A binary arithmetic coder is used for the special case when the source data to be encoded is made up of symbols from a binary alphabet i.e. having only two symbols, 0 and 1.

The implementation by Moffat *et al.* provides a binary arithmetic coder, which is a modification of the multisymbol arithmetic coder given in the previous section. The pseudocode is shown in Figure 3.4. A number of the components in the multisymbol arithmetic coder were eliminated:

1. There is no need for a data structure to keep the statistics since cumulative frequency counts are trivially available. Since the frequency counts of only two symbols, 0 and 1, denoted by c_0 and c_1 respectively, are kept, the total count is found by adding them.
2. The term MPS (more probable symbol) is used to denote the symbol having a probability of 0.5 or more. The other symbol is denoted as LPS (less probable symbol). As mentioned in Section 3.3.2, the MPS should be maintained at the top of the probability model. This was done by rearranging the code.
3. One multiplicative operation was eliminated in the encoder when LPS is transmitted, and one multiplicative operation avoided in the decoder for the MPS * and two for the LPS.

```

function binary_arithmetic_encode(c0, c1, bit)


---


if (c0 < c1){
    LPS ← 0
    cLPS ← c0
}else{
    LPS ← 1
    cLPS ← c1
}
r ← R/(c0+c1)
rLPS ← r*cLPS

if (bit = LPS){
    L ← L+R-rLPS
    R ← rLPS
}else
    R ← R-rLPS

renormalize as in multisymbol arithmetic encoder

function binary_arithmetic_decode(c0, c1)


---


if (c0 < c1){
    LPS ← 0
    cLPS ← c0
}else{
    LPS ← 1
    cLPS ← c1
}
r ← R/(c0+c1)
rLPS ← r*cLPS

if D ≥ (R-rLPS){
    bit ← LPS
    D ← D-(R-rLPS)
    R ← rLPS
}else{
    bit ← 1-LPS
    R ← R-rLPS
}
renormalize as in multisymbol arithmetic decoder
return bit

```

Figure 3.4: Pseudocode of Moffat *et al.*'s binary arithmetic coder

3.4 Modeling

Figure 3.5 shows how the arithmetic coder is incorporated into a compression system based on the general image compression model given in Figure 2.2 of Section 2.6. As this dissertation uses lossless compression, the quantizer block is deleted.

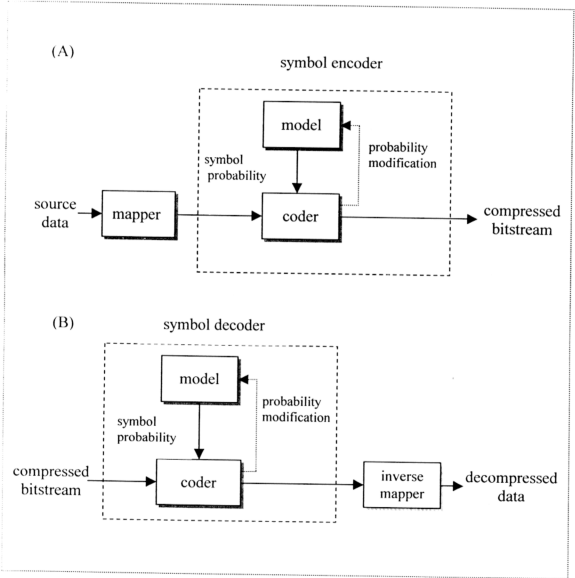


Figure 3.5: Incorporating arithmetic coding into a compression system

The arithmetic coder is divided into two independent modules, the model and coder, as advocated by Rissanen & Langdon (1981). The model provides an estimate of the symbols' probabilities i.e. it models the statistics of the data. The coder (as described in the previous sections) encodes a symbol based on its probability, as provided by the model, to generate a sequence of bits.

For a given model, arithmetic coding will give close to optimum compression (as mentioned in Section 2.7). As suggested by Moffat *et al.* (1998), the model can be considered as the “intelligence” of the compression system, whereas the coder is the “engine room”. Improvements to the model will yield improvement to the compression effectiveness, in terms of reduced size of the compressed data. Improvements to the coder are mainly concerned with the compression efficiency, that is, a reduction in time and memory usage.

Because of this, the focus of this dissertation is on implementation of the model, rather than the coder. As mentioned in the previous section, the coder uses a readily available implementation of the improved arithmetic coder by Moffat *et al.* (1998).

There are three types of modeling (Langdon, 1981; Howard & Vitter, 1994):

1. *Static or non-adaptive model*

A predefined model is fixed and used for compressing all data sources. It is obtained from measuring some sample data or making certain assumptions. This would work well if the probability distribution of all the data exactly matches the fixed model. In the case of images, this may not happen because different images may have different probability distributions. For example, one image may have the colour red appear most often, whereas in another image the colour blue appears most often.

2. *Semi-adaptive model*

Two passes are made over the data. In the first pass, statistics are collected to build the exact model. In the second pass, the model is used to compress the data. The model is sent along with the compressed data to the decoder. This incurs an overhead, especially if the size of the model is large. Also, the whole data needs to be known before compression can commence.

3. *Adaptive model*

The model is built on-the-fly while the data is compressed, based on the data already seen by the coder. The system ‘learns’ the statistics of the data during the compression process and can adapt to the probability distribution of different images. Unlike the semi-adaptive model, there is no need to send the model to the decoder because the decoder imitates the encoder when building up the model. It also allows for incremental compression of the data without having to wait for the rest of the data to be known.

It has also been shown that the static model can be arbitrarily bad, while, in general, an adaptive model performs as well as the semi-adaptive model (Howard & Vitter, 1994). Therefore, the adaptive model is used in this dissertation.

Figure 3.5 shows the role of the adaptive model in the compression system. The symbol to be encoded is input to the coder. The symbol’s probability is obtained from the model. After encoding the symbol, the model will be updated, thus building a model of the data’s statistics. At the decoder, the compressed bitstream is decoded according to the model. The decoder builds the model adaptively in exactly the same manner as the encoder, thus ensuring decodability.

3.5 Context-based statistics modeling

One approach to modeling is to use the context in which the symbol to be encoded appears in to determine its probability (Salomon, 2000; Langdon & Rissanen, 1981). The context consists of symbols that have been encountered before the current symbol. The order of the model refers to the number of previous symbols c that makes up the context. An order 0 model means that the probability of the symbol is independent of any previous symbols. An order c model means that the probability of the symbol depends on c previous symbols.

The following example shows intuitively how using context modeling can be effective. Consider compression of an English language text. The letters ELEPHAN are seen preceding the next letter to be encoded. Then, we can say almost for certain that the next letter is T because ELEPHANT is the only valid word in the English language. There is a very high probability that the next letter is T and the number of bits required to encode the letter should be very small. The letter T has dependency on the previous letters, and based on this dependency it is highly redundant (i.e. even without the letter T, we can predict the word will be ELEPHANT).

This can be extended to the case of an image. An image has spatial dependencies between pixels that can be exploited. The pixel to be encoded is expected to be correlated with its neighbours. Therefore, the neighbouring pixels can be used to predict the probability of the pixel.

The model is a probability distribution of the symbols. It keeps a list of symbols and their corresponding probabilities. When encoding a symbol, its probability will be supplied to the arithmetic coder.

In an actual implementation, the frequency count of the symbol is kept instead. It is not possible to find the ‘real’ probability since in reality the true statistical nature is an enigma (Withers, 2001). The frequency count of the symbol is the number of times a symbol has appeared in that context. The frequency count is used as an estimate of the symbol’s probability (using equation 3.5). In the adaptive model, the frequency count is modified when a symbol is being encoded, thereby updating the model.

3.5.1 Neighbourhood template

For an image, the context is found based on a *neighbourhood template*. The neighbourhood template consists of selected pixels in the vicinity of the pixel being encoded. Let s be the size of the alphabet and c the number of pixels forming the template. The number of possible contexts is s^c .

The values of the pixels can be interpreted as a distinct index value pointing to a statistical model. Each model is a table (or some other form of data structures) that stores the symbol probabilities. The *order* of the model is equal to the number of pixels forming the template.

In practice, the neighbourhood pixels used must be *causal* pixels, that is, they have been previously seen by the coder when encoding the current pixel. The reason is because during decoding, only the pixels that have been decoded will be known to the decoder, thus enabling to build the model exactly in step with the encoder.

Note that there are cases when the template or a portion of it lies outside the image. In this case, a default value of 0 is used for the pixels that are outside the image boundary.

Figure 3.6 shows an example of a four-pixel neighbourhood template of a binary image. In this case, the number of possible contexts is $2^4 = 16$. The pixel denoted X is the current pixel being encoded. Four neighbouring pixels are chosen to form the template, their values as shown. The index value can be interpreted as, for example, 1101_2 or 13_{10} (the ordering is not important as long as it is consistently obeyed). This is then used to find the corresponding statistical model from a table.

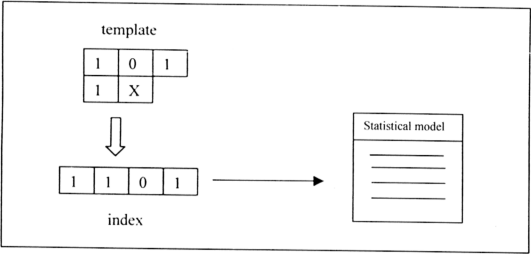


Figure 3.6: An example of a neighbourhood template

The selection of the pixels forming the neighbourhood template affect the accuracy of the model, hence the compression performance (Ageenko *et al.*, 2001). In this dissertation, the following templates were used:

1. Standard 1-norm and 2-norm templates (Martins & Forchhammer, 1998)
2. Templates used by JBIG2 (JBIG committee, 1999)

Figure 3.7 shows the standard 1-norm and 2-norm templates, where X is the current pixel. For example, a 12-bit 1-norm template is made up of the pixels marked 1 to 12. Figure 3.8 shows the templates defined for use in the JBIG2 standard. In the JBIG2 standard, a template may have adaptive pixels whose locations are flexible. Here, the

adaptive pixels are placed at their nominal locations. Note that one of the 10-bit template (C) is the same as that obtained from the 2-norm template.

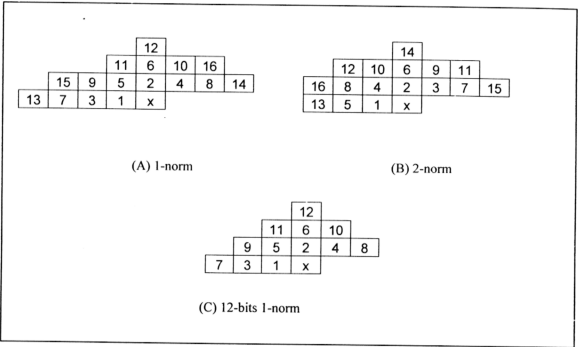


Figure 3.7: 1-norm and 2-norm neighbourhood templates

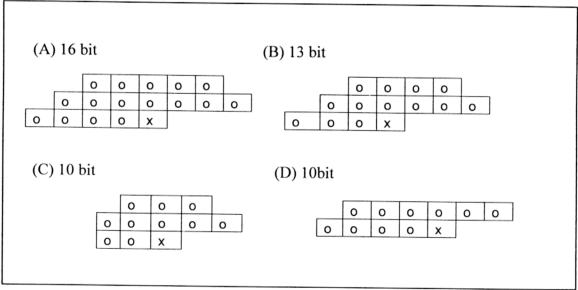


Figure 3.8: Neighbourhood templates used by JBIG2 standard

Two methods involving context-based statistics modeling are bitplane coding and Prediction by Partial Matching (PPM). They will be described next.

3.6 Bitplane coding

Consider an image of the size $H \times W$, where H and W denote the height and width of the image respectively. Let p be the number of bits used to represent the image's pixel value. By selecting a single bit from the same position in the binary representation of each pixel, an $H \times W$ image, called a *bitplane*, can be formed. It is a binary image consisting of only the values 0 and 1. The original image can thus be decomposed into a set of p $H \times W$ binary image.

As a convention, the *bitplanes* are numbered 0 through $p-1$. Bitplane 0 represents the binary image formed by taking the most significant bit of each pixel. The next bitplane represents the binary image formed by taking the next most significant bit of each pixel, and so on, up to bitplane $p-1$ which represents the binary image formed by taking the least significant bit of each pixel. Figure 3.9 shows an example of the decomposition of an 8 bits-per-pixel image into the 8 separate bitplanes.

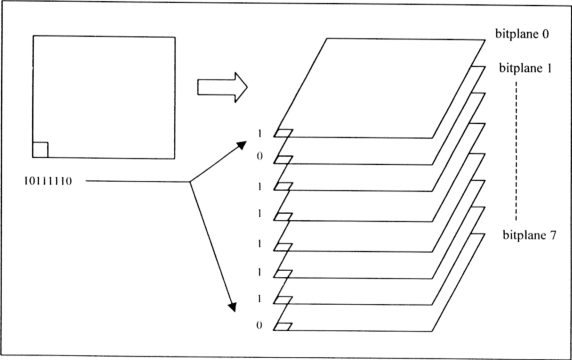


Figure 3.9: Bitplane decomposition

A binary arithmetic coder is then used to compress each bitplane, i.e. each bitplane is treated as a binary image. Context modeling is performed independently on each bitplane to estimate the symbols' probabilities using neighbourhood templates. The following templates were investigated:

1. Standard 1-norm and 2-norm templates
2. Templates used by JBIG2, with the adaptive pixels at their nominal locations

For the standard 1-norm and 2-norm templates, the template size (i.e. the number of bits forming the context) was varied to find the most effective compression.

Two additional factors affecting compression that were investigated are bitplane reduction and the use of reflected Gray codes.

3.6.1 Bitplane reduction

Bitplane reduction is a preprocessing step that can be used to improve the compression of bitplane coding (Yoo *et al.*, 1998). In the raw data to be compressed, a pixel value may be represented by more bits than actually required. For example, let's take the case of 8 bits used for each pixel, where up to 256 unique colours can be represented. The value of the pixels may be anything in the range of $[0,256)$. In the actual map image, significantly less number of colours may be used, for example 20 colours.

Therefore, bitplane reduction ensures that only just enough number of bits is used to represent the pixel values, regardless of how many bits are used in the original raw data, so that improved compression can be achieved. For example, in the case of an image having 20 unique colours, 5 bits per pixel is sufficient. Thus, only five bitplanes need to be encoded. Any extra bitplanes are redundant, being made up of all zeros.

Bitplane reduction is performed by the encoder as a preprocessing step (the mapper block in Figure 3.5), before decomposing the bitplanes. First the number of unique colours that is used in the image, s , is counted. The pixel integer values in the range $[0, 2^p)$ are mapped to integer values in the range $[0, s)$, where p is the original number of bits per pixel. The assignment of the new value follows the order of appearance in the image in a raster scan, incrementally, starting from the value 0. The reduced number of bits per pixel can then easily be found from the value of s , as shown in Table 3.7.

Table 3.7: Minimal bits per pixel required

number of unique colours	bits per pixel
1-2	1
3-4	2
5-8	3
8-16	4
17-32	5
33-64	6
65-128	7
129-256	8

Figure 3.10 shows an example of the bitplane reduction process. Starting from the top row and the leftmost column, the first value to appear is 212. It is assigned the new value of 0. The second value is 23, and it is a new value, so it is assigned 1. The next new value to appear is 128, which is assigned 2, and so on. The resultant image has the values 0 to 4, sufficiently represented by 3 bits per pixel, so only 3 bitplanes need to be encoded.


At the decoder side, the inverse of bitplane reduction will also have to be performed after the data is decompressed, so that the original data is obtained.

(A) original

212	23	23	23	128	128	128	128
212	0	0	0	0	0	0	128
212	0	0	0	0	0	0	128
212	23	23	23	23	23	0	128
212	0	0	0	0	0	0	128
212	1	1	1	1	1	0	128
212	1	1	1	1	1	0	128
212	128	128	128	128	128	128	128

(B) bitplane reduced image

0	1	1	1	2	2	2	2
0	3	3	3	3	3	3	2
0	3	3	3	3	3	3	2
0	1	1	1	1	1	3	2
0	3	3	3	3	3	3	2
0	4	4	4	4	4	3	2
0	4	4	4	4	4	3	2
0	2	2	2	2	2	2	2



translation table	
original	new
212	0
23	1
128	2
0	3
1	4




Figure 3.10: An example of the bitplane reduction process

3.6.2 Reflected Gray coding

A possible improvement is to map the pixel values from its normal binary representation to reflected Gray code before compression. In the case of continuous-tone grayscale images, this brings some improvement to the compression performance (Rabbani & Melnychuck, 1992). Therefore, the use of reflected Gray codes was investigated to see whether it would improve compression compared to using normal binary representation.

In reflected Gray codes, the consecutive integers differ by one bit only. Table 3.8 shows the reflected Gray codes for the integers 0 to 15. The reflected Gray code can be found

from the binary representation using an exclusive-OR operation, as in the following equation:

$$g(i) = b(i) \text{ XOR } b(i+1) \tag{3.14}$$

where

$g(i)$ is the i -th bit in the reflected Gray code representation

$b(i)$ is the i -th bit in the binary code representation

To find the binary code from the reflected Gray code, the same equation can be used.

Figure 3.11 shows an example of converting between binary code and Gray code.

Table 3.8: Reflected Gray codes for 0 to 15

Decimal	Binary code	Gray code	Decimal	Binary code	Gray code
0	0000	0000	8	1000	1100
1	0001	0001	9	1001	1101
2	0010	0011	10	1010	1111
3	0011	0010	11	1011	1110
4	0100	0110	12	1100	1010
5	0101	0111	13	1101	1011
6	0110	0101	14	1110	1001
7	0111	0100	15	1111	1000

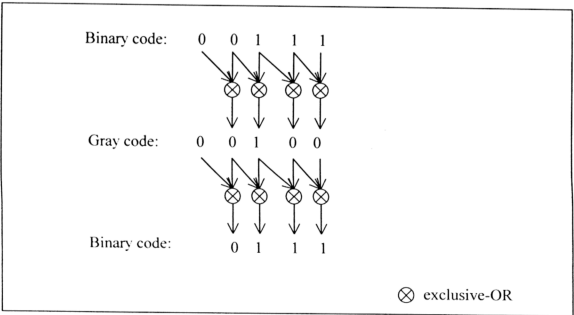


Figure 3.11: Converting between binary code and Gray code

3.7 Prediction by Partial Matching (PPM)

Prediction by Partial Matching (PPM) uses variable length context modeling instead of fixed length context modeling (Cleary & Witten, 1984; Moffat, 1990; Howard & Vitter, 1992). The idea is that, the higher the order of the model, the better the compression. However, during the early stages of compressing the data, the statistics of a higher order model takes some time to become accurate, hence the compression is relatively ineffective (Cleary & Witten, 1984). The solution used by PPM is that the lower order models are used if the higher order models are not yet available.

It starts by attempting to encode a symbol using the largest order model, where the size of the largest order is some predetermined value. If the symbol to be encoded has not been encountered previously in that context (a *novel* symbol), an *escape* symbol is encoded. The algorithm then attempts to use the next smaller order model to encode the symbol.

This process is repeated, resulting in two possibilities:

1. A context that has been encountered with the symbol is obtained. The symbol can then be encoded using the model associated with that context. The arithmetic coder is invoked to encode that symbol
2. The symbol has not been encountered previously in any context, including order 0. In this case, the symbol is encoded using a default model, called the order -1 model. This is an equiprobable model where all symbols of the alphabet are given the frequency count of 1. Thus the symbol is encoded by the arithmetic coder with the probability of $1/s$, where s is the size of the source alphabet.

Several factors affecting compression and which were investigated are described next.

3.7.1 Maximum order

The maximum order is the size of the largest order model which the method starts with when attempting to encode a symbol. There is an optimal value for the maximum order at which compression effectiveness is the best and this value depends on the type of data being compressed (Cleary & Witten, 1984). Therefore, several values of maximum order were experimented with to find the optimal value.

3.7.2 Escape probability

The first time a symbol is encountered for a particular context, its frequency count is zero. This is known as the *zero frequency problem*. An escape symbol needs to be encoded with a non-zero probability. What probability is assigned has been found to affect the compression performance (Cleary & Witten, 1984; Moffat, 1990; Howard & Vitter, 1992). There are however no theoretical basis for assigning the escape symbol probability, and the use of heuristics are relied upon instead. Several methods have been proposed.

The following methods, known simply as A, B, C and D were implemented:

1. Method A – the escape symbol is allocated the frequency count of 1. The total count for the context is inflated by 1 (Cleary & Witten, 1984).
2. Method B – all the distinct symbols that have been encountered in the context have their frequency count subtracted by one. The total obtained from this is assigned to the frequency count of the escape symbol. Thus the total count of the context is preserved (Cleary & Witten, 1984).
3. Method C – the escape symbol's count equals the number of distinct symbols that have been encountered in the current context. The total count is inflated by the same amount (Moffat, 1990).

4. Method D – each time a novel symbol is encountered, its count is assigned the value of $\frac{1}{2}$, while the other $\frac{1}{2}$ is assigned to the escape symbol. Thus the escape symbol's count is $d/2$, where d is the number of distinct symbols that have been encountered in the current context. The total count is inflated by 1 (Howard & Vitter, 1992).

Table 3.9 summarizes the escape and symbol probabilities used by these methods. The following notations are used:

- p_{esc} = probability of escape symbol
- p_s = probability of symbol being encoded
- t = total frequency count
- f = frequency count of symbol being encoded
- d = number of distinct symbols encountered in the current context

Table 3.9: Escape and symbol probabilities

	A	B	C	D
p_{esc}	$1 / (t + 1)$	d / t	$d / (t + d)$	$d / 2t$
p_s	$f / (t + 1)$	$(f - 1) / t$	$f / (t + d)$	$(f - \frac{1}{2}) / t$

As a note, the escape method used is often used to define the type of PPM method and distinguishes it from others. PPM implementations employing these escape methods are also called as PPMA, PPMB, PPMC, PPMD etc.

3.7.3 Exclusion principle

When switching down from a larger order to a smaller order, symbols that have already been seen in the previous order can be excluded when finding the symbol probabilities of the smaller order that need to be passed to the arithmetic coder. The increase in the smaller order's symbol probabilities improves compression (an example of this can be

found in Moffat, 1990). The exclusion principle was applied in the PPM implementation.

3.7.4 Update exclusion

This is an improvement to PPM proposed by Moffat (1990). Each time a non-escape symbol is encoded, its frequency count in the probability models for the associated context are incremented. In *full counting*, the probability model of all the contexts from the largest order down to order 0 is updated. In *single counting* (also called *update exclusion*), only the models for the context at or above the order in which it was successfully encoded are updated.

For example, let the largest order be 4. A symbol is successfully encoded at order 2. In the case of single counting, the probability models of the context for order 4, 3 and 2 are updated. In full counting, the probability models of the context for all the order 4, 3, 2, 1 and 0 are updated.

3.7.5 Frequency count scaling

Practical implementations of the arithmetic coder require some limit to be imposed on the total frequency count of the symbols. This is done because of the limitation imposed by the size of the variable storing the value of the total frequency count. For example, if the variable is 16 bits, then the limit on the total count would be $2^{16}-1$, i.e. 65535. When the limit is reached, all of the counts of the probability model in that particular context are halved.

The side effect of this is that it refreshes the statistics of the model and makes it self-adapting to the changing symbol distributions in the input data (Moffat, 1990). For

example, if a certain colour tends to appear often only in an early part of the image, then the adaptive model will reflect this by assigning it a high frequency count. However, at a later part of the image, when the colour no longer appears as often, the statistics are no longer accurate. Using scaling will help the model adapt to this change more quickly. Different values of the maximum frequency count were experimented with to find its effect on compression performance.

3.7.6 Neighbourhood template

As in the case of bitplane coding, a neighbourhood template is used to determine the context. The standard 1-norm and 2-norm templates were investigated.

3.8 Combined method of bitplane coding and PPM

In the combined method, both bitplane coding and PPM, as described in the previous sections, are used. They each operate independently to compress different planes of an image.

The original image planes are separated into two parts:

1. The 4 LSB bitplanes, treated as a combined, single individual plane of m -ary values, where $m \leq 16$, i.e. the plane consists of symbols from the set $\{0, 1, \dots, m-1\}$.
2. The rest of the upper bitplanes, if any, each consisting of binary symbols

PPM is used to compress the 4 LSB bitplanes as a single combined plane, where the number of symbols can be up to 16. Bitplane coding is used to compress the rest of the bitplanes individually as binary images. If the number of bitplanes in the original image is 4 or less, only PPM is used. Figure 3.12 shows an example.

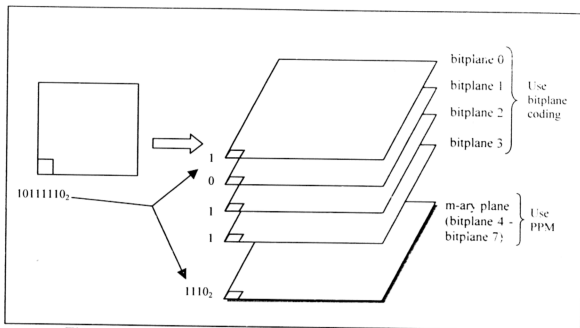


Figure 3.12: An example of splitting an image into different planes

3.9 Test images

To test and compare the compression methods, a set of test images were used. In data compression research, the performance of a compression method is usually tested on a collection of test images or predefined files known as a *corpus*. The purpose of establishing a corpus is so that the performance of various compression methods can be compared against each other based on a common set of source data. Two well-known examples are the Canterbury and Calgary corpus. The Canterbury corpus is a set of text files. The Calgary corpus contains various types of files (e.g. image, text, program source code, executable file).

Since the focus of this dissertation is on compressing road map images, a road map corpus was established. Figure 3.13 depicts the procedure. The steps taken were:

1. Fifteen maps were obtained from several websites. These maps are of various sizes and number of colours used. The map images are included in Appendix A.