Chapter 4:

Implementation

4. Implementation

This chapter elaborates in detail the implementation of the compression programs. Pseudocode and snippets of C code will be presented where appropriate to clarify the descriptions. Section 4.1 considers the general structure of the implemented programs. Section 4.2 and 4.3 describes bitplane coding and PPM respectively. Section 4.4 describes an implementation that combines both.

4.1 General structure

The compression program reads the raw image data from a file, compresses the data and writes the resulting bitstream to an output file. The decompression program does the opposite. It reads the bitstream from the compressed file, decompresses it and writes the reconstructed raw image data to a file.

The general structure of the compression and decompression program is shown in Figure 4.1 and 4.2 respectively. The figures show the blocks involved and the flow of data. The blocks roughly correspond to the general image compression system shown in Figure 3.5, as indicated at the right of the figures.

The function of each block in the compression program is as follows:

1. Front-end – it interfaces with the user to obtain the file's information and compression parameters. To make the testing process more convenient, this is done using a script file, so the user can list down all the files to be compressed in the script file along with the parameters and then let the program run. It then reads the raw image data from the source file, performs the required preprocessing and passes the data to the file-level encoder.

2. File-level encoder – it determines which symbol is to be encoded next and the context for that symbol, which are both passed to the symbol-level encoder. It also writes the header information to the output file before compression begins and performs initialization/termination of the arithmetic coder.
3. Symbol-level encoder – it finds the probabilities required by the arithmetic coder to encode the symbol. It also manages the model's statistics.
4. Arithmetic encoder – it performs compression by encoding a symbol based on the probabilities passed to it. It outputs the compressed data as a bitstream which is written to an output file.

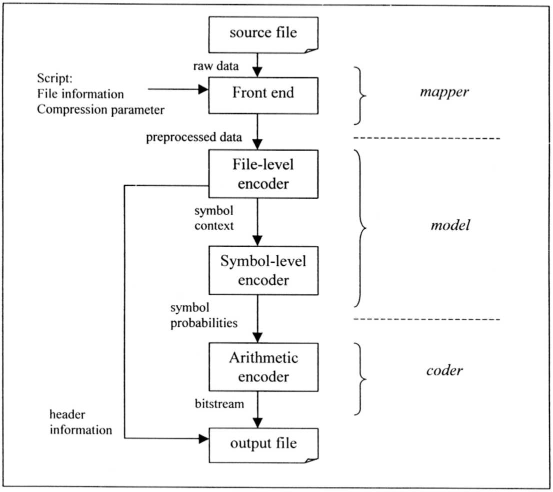


Figure 4.1: General structure of the compression program

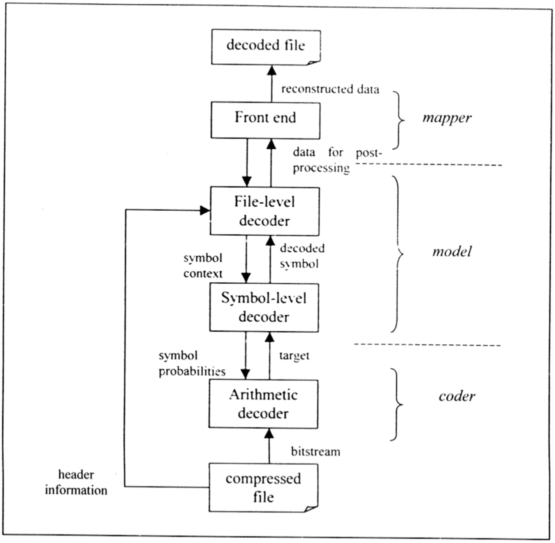


Figure 4.2: General structure of the decompression program

The function of each block in the decompression program is as follows:

1. Front-end – it receives the decoded data, performs the required post processing and writes the reconstructed data to a file
2. File-level decoder – it determines the context and passes it to the symbol-level decoder, which will return the decoded symbol. It is also responsible for reading the header information from the compressed file compression and performs initialization/termination of the arithmetic coder.
3. Symbol-level decoder – it finds the probabilities required by the arithmetic decoder to decode a symbol. The arithmetic decoder will return a target value, from which this block will determine the decoded symbol. It also manages the model's statistics.

4. Arithmetic decoder – it reads the compressed bitstream from the file and finds the target value based on the probabilities passed to it.

As mentioned previously, the improved arithmetic coder implementation of Moffat *et al.* (1998) was used as the compression "engine" of the programs. The following functions related to the arithmetic coder were used:

1. `arithmetic_encode( )`
2. `arithmetic_decode_target( )`
3. `arithmetic_decode( )`
4. `binary_arithmetic_encode( )`
5. `binary_arithmetic_decode( )`
6. `start_encode( )`
7. `finish_encode( )`
8. `start_decode( )`
9. `finish_decode( )`
10. `startinputtingbits( )`
11. `startoutputtingbits( )`
12. `doneinputtingbits( )`
13. `doneoutputtingbits( )`

Detailed explanation of these functions will not be given here but can be found in their paper. The pseudocodes for functions (1)-(9) were already given in Section 3.3. Functions (10)-(13) are initialization and termination functions for file data input and output and will be described briefly where appropriate.

4.2 Bitplane coding

The compression program is described followed by the decompression program, based on the general structure mentioned above.

4.2.1 Compression program

Implementation of the front end, file-level encoder and symbol-level encoder is described next.

4.2.1.1 Front end

The flow for the front end of the compression program for bitplane coding is given in Figure 4.3.

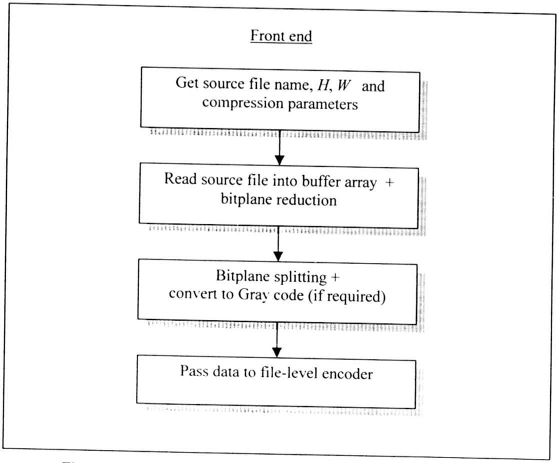

Front end
Get source file name, *H*, *W* and compression parameters
Read source file into buffer array + bitplane reduction
Bitplane splitting + convert to Gray code (if required)
Pass data to file-level encoder

Figure 4.3: Front end of compression program for bitplane coding

First, the program obtains from the user the name of the file to compress and the image's height, *H* and width, *W*. For experimenting with the compression parameters, the program should allow the parameters to be changed. For bitplane coding, the parameters are:

i. whether to use reflected Gray code or not

ii. the template type

For item (ii), different versions of the program were compiled. Item (i) was made as a run-time option.

Next, the data from the file is read into a buffer array. It is assumed that each byte of the file's data is an individual symbol to be encoded. Each byte is read serially, which is equivalent to scanning the image in a raster fashion, from left-to-write, top-to-bottom.

The following preprocessing steps are performed on the data:

1. Bitplane reduction
2. Bitplane splitting
3. Conversion into reflected Gray code, if required

4.2.1.1.1 Bitplane reduction

As explained in Section 3.6.1, bitplane reduction is applied to ensure that only just enough bits are used to represent the pixel values, so that optimum compression can be achieved. Figure 4.4 shows the pseudocode for the bitplane reduction process, which is performed on-the-fly while reading the data from file to a buffer array.

```
initialize orig_new[ ] elements to -1
s ← 0
for (i=0 to (HxW)-1){
    temp ← byte read from source file
    if (orig_new[temp] = -1){
        orig_new[temp]←s
        new_orig[s]←temp
        s ← s+1
    }
   buffer[i]← orig_new[temp]
}

assign value to bpp according to value of s:
  0 < s ≤ 2   : bpp ← 1
  2 < s ≤ 4   : bpp ← 2
  4 < s ≤ 8   : bpp ← 3
  8 < s ≤ 16  : bpp ← 4
  16< s ≤ 32  : bpp ← 5
  32< s ≤ 64  : bpp ← 6
  64< s ≤ 128 : bpp ← 7
  128< s ≤ 256: bpp ← 8
```

Figure 4.4: Pseudocode for bitplane reduction process

A translation table is set up so that the decompression program can do an inverse mapping to recover the original values. The array `orig_new[]` maps the original value to the new value. The index of the array represents the old value, while the corresponding array element is the assigned new value. The array `new_orig[]` does the opposite. It maps the new value to the original value. The index of the array represents the new value, while the corresponding array element is the original value.

The elements of the array `orig_new[]` are initialized to -1, to indicate that the corresponding symbol have not been seen before in the process. The variable `s`, used for keeping track of the number of unique colours (i.e. symbols) seen so far, is initialized to zero.

Each symbol is read from the file into a variable, `temp`. The array `orig_new[]` is consulted to check whether the symbol has been seen before. If not, then the arrays orig_new[]and new_orig[] are modified to register the new symbol, and the variable s is incremented. Finally, the buffer array is filled with the equivalent bitplane-reduced value of the symbol read from the file.

At the end of the process, s gives the number of different symbols seen i.e. the number of unique colours used in the image. The reduced number of bits per pixel, bpp, is found from the value of s.

4.2.1.1.2 Bitplane splitting

Bitplane splitting decomposes the *H*x*W* image data into *p HxW* bitplanes, where *p* is the number of bits per pixel. Figure 4.5 shows the pseudocode for the process. The conversion of the symbols from binary code to reflected Gray code representation is done if required. It is integrated into the same process.

The array data[] is used to store the data after bitplane splitting. The variable use_gc is used to indicate whether to apply Gray coding.

```
if (use Gray code)
   use_gc ← 1₂
else
   use_gc ← 0₂

for (i=0 to (HxW)-1){
  temp ← buffer[i]
  bit_number ← bpp
  for (p=0 to bpp-1){
     bit_number ← bit_number-1
     bit ← temp >> bit_number
     data[p*H*W+i] ← (bit XOR(use_gc AND (bit>>1))) AND 1₂
   }
}
```

Figure 4.5: Pseudocode for bitplane splitting

Each symbol is read from the buffer. For each bitplane, the relevant bit is found by extracting it from the symbol using a shift operation. The particular bit, contained in the variable `bit`, is right-shifted to the LSB position by an amount indicated by the variable `bit_number`. Conversion to reflected Gray codes is also performed if required, based on the equation described in Section 3.6.2. The variable `bit` is shifted one bit to the left and bitwise-exclusive-ORed with its original value. Finally, the bit is extracted using a bitwise-AND operation with the mask 00000001_2.

4.2.1.2 File-level encoder

The file-level encoder is responsible for encoding of the file as a whole. The flow of the function is given in Figure 4.6.

Before starting the compression, relevant information are written as the header of the output file. A couple of initialization functions, `startoutputtingbits( )` and `start_encode( )`, are called. Then, the models are created and initialized.

To encode the symbols in the bitplanes, the context of each symbol is found and passed along with the symbol to the symbol-level encoder. When all the symbols have been encoded, the termination functions, `finish_encode( )` and `doneoutputtingbits( )`, are called.

The next sections elaborate on these steps.

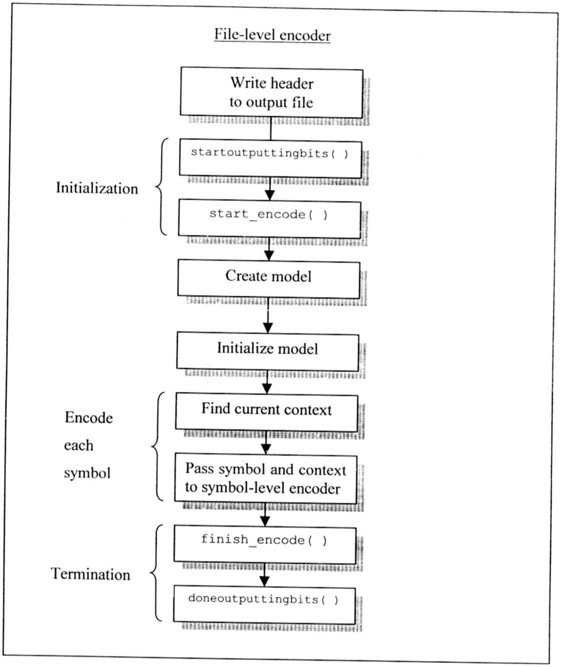


Figure 4.6: File-level encoder for bitplane coding

4.2.1.2.1 File header

The organization and contents of the file header will very much depend on the actual software implementation. At the minimum, the information required includes the image's height, width, the number of symbols and the translation table created by the bitplane reduction process. Also, the palette table needs to be included since it is likely that different files will use different palette tables. If the final software implementation allows the user to select the compression parameters, then these options will have to be included in the file header so that the decoder knows which option was used by the encoder.

4.2.1.2.2 Initialization

The function `startoutputtingbits( )` initializes an output buffer to 0. The output buffer is a one byte variable used to store the bits generated by the arithmetic coder during the compression process. The arithmetic coder generates the compressed bitstream one bit at a time. Since it is to be stored in a file, the bitstream needs to be written to the file on a byte-by-byte basis. This is done by storing the bits temporarily in the output buffer. When the buffer has collected enough bits to become a full byte, it is written to the file and the buffer is cleared.

The function `start_encode( )` initializes the arithmetic coder's range variables, L and R, such that the coding range is set at $[0, 2^{b-1})$, as required by the arithmetic coder.

4.2.1.2.3 Creation and initialization of the models

There are three things that need to be done with regards to the models:

1. Creating the models
2. Initializing the models
3. Managing the models
    a. updating the models' statistics when a symbol is encoded
    b. scaling down when applicable

Item (3) is performed by the symbol-level encoder. As such, only item (1) and (2) is described here.

Creating the models

Before encoding can commence, the models are created and initialized. Each model is a probability table containing statistics of the data being encoded. There is a model associated with each possible context. A C *struct* type is used to store the probability table, as shown by the C code snippet below:

```
typedef struct {
    unsigned short c0;
    unsigned short c1;
} bpc_prob_table;
```

The variable `c0` stores the frequency count for the symbol 0, while `c1` stores the frequency count for the symbol 1.

The models are stored in an array whose size is given by the number of possible contexts, 2^c, where c is the number of context bits. Each element of the array is the type `bpc_prob_table`. The array is dynamically allocated during run-time. In practice, if the number of context bits is fixed, the array size can also be fixed during compile-time. Here dynamic allocation was used because of the need to experiment with various number of context bits (i.e. different template types).

Initializing the models

To initialize the models, `c0` and `c1` of each probability table are assigned the value of 1. The models are reinitialized each time a new bitplane is being encoded. This is to ensure fresh statistics are used for each bitplane, i.e. statistics from previous bitplane are not used for the next bitplane.

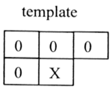
4.2.1.2.4 Finding the context

The context for a symbol is found using a neighbourhood template. The template consists of a set of pixels in the neighborhood of the current pixel that is being encoded. The context is calculated from the pixel values as a binary digit, which then serves as an index to the array containing the corresponding probability table.

In the implementation, C macro statements are used to find the context, as shown in an example in Figure 4.7. The *conditional if* is used to test whether a template pixel lies outside the image boundary. If yes, then the pixel is assigned the value 0. The shift operation and bitwise-OR are used to move the context bits to their appropriate positions. To find the context, the macro statement is called, e.g.:

```
context = TEMPLATE(data);
```

where `data` is a pointer to the data buffer array.



```
#define BIT_W(x)      column==0?0:*(x+ptr-1)
#define BIT_N(x)      row==0?0:*(x+ptr-W)
#define BIT_NE(x)     (column==W-1||row<1)?0:*(x+ptr-W+1)
#define BIT_NW(x)     (column<1||row<1)?0:*(x+ptr-W-1)

#define  TEMPLATE(x)
(BIT_W(x))|(BIT_N(x)<<1)|(BIT_NE(x)<<2)|(BIT_NW(x)<<3)
```

Figure 4.7: Example of a code snippet used to find the context

This approach was used to find the context since it is easy to modify, hence suitable for testing purposes when we want to experiment with various template types and sizes. In an actual implementation, if the template and maximum order is fixed, it would be more efficient to embed the process of calculating the context into the encoding loop of Figure 4.8 without the need to test whether a neighbour pixel is outside the image boundary.

4.2.1.2.5 Encoding each symbol

For each symbol, the symbol and its context are passed to the symbol-level encoder. Encoding is done plane-by-plane, starting from the MSB bitplane down to the LSB bitplane, as shown in the pseudocode given in Figure 4.8. Before encoding begins for each new bitplane, the probability tables are reinitialized, as mentioned in the previous section.

```
ptr ← 0
for (p=0 to bpp-1){
   initialize models
   for (row = 0 to H-1)
         for (column = 0 to W-1){
            symbol ← data[ptr]
            find the context for symbol
            call symbol-level encoder
            ptr ← ptr+1
         }
}
```

Figure 4.8: Pseudocode for encoding each symbol

4.2.1.2.6 Termination

The function `finish_encode( )` is called to terminate the arithmetic coder. It involves calculating and outputting sufficiently many bits so that the final interval can be identified by the decoder unambiguously, irrespective of what other bits follow on from there (Moffat *et al.*, 1998).

The function `done_outputting_bits( )` function writes the final byte to the file. This happens if the output buffer byte has not yet been completely filled to 8 bits. In this case, the buffer will be shifted so that the bits occupy their appropriate positions and then the byte is written to the file.

4.2.1.3 Symbol-level encoder

The symbol-level encoder is responsible for encoding a symbol by calling the arithmetic coder and managing the model, as shown in Figure 4.9. First the symbol is encoded by calling `binary_arithmetic_encode( )`. Then the probability table for the current context is updated and the frequency counts scaled down if necessary.

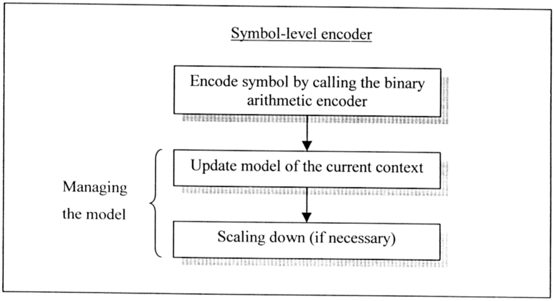

Symbol-level encoder
Encode symbol by calling the binary arithmetic encoder
Update model of the current context
Scaling down (if necessary)
Managing the model

Figure 4.9: Symbol-level encoder for bitplane coding

Managing the models

Figure 4.10 shows the pseudocode for managing the models. The models are updated each time a symbol encoded, in order to reflect the latest statistics, as per the requirement of adaptive modeling. The frequency count of the symbol being encoded in the current context's model is incremented.

Scaling down the frequency counts is also performed when a defined maximum value is exceeded. This has to be done in order to prevent overflow of the variables `c0` and `c1`. The values of `c0` and `c1` are halved to the smallest integer equal or larger than the halved value.

```
do for the current context's probability table:

/*update the probability table*/
if (symbol being encoded = 0)
   c0 ← c0+1
else
   c1 ← c1+1

/*scale down if necessary*/
if (c0 + c1 > maximum value){
      c0 ← (c0+1) >> 1
      c1 ← (c1+1) >> 1
}
```

Figure 4.10: Pseudocode for model management

4.2.2 Decompression program

Implementation of the decompression program is described next. Some functions are identical to those described in the previous sections. In this case, their description will not be repeated and the reader is kindly requested to refer to the relevant section.

4.2.2.1 Front end

The flow of the front end is given in Figure 4.11.

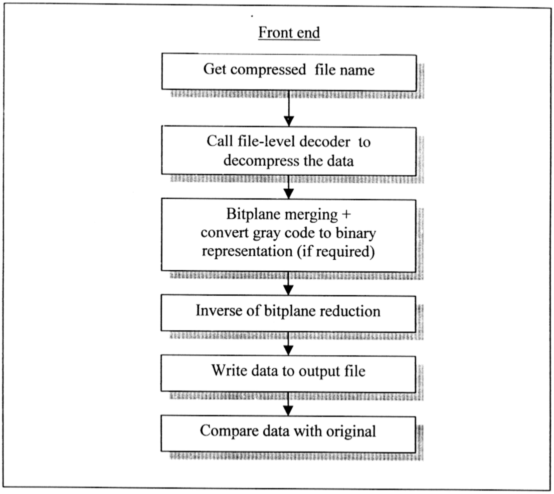

Front end
Get compressed file name
Call file-level decoder to decompress the data
Bitplane merging + convert gray code to binary representation (if required)
Inverse of bitplane reduction
Write data to output file
Compare data with original

Figure 4.11: Front end of the decompression program for bitplane coding

The program obtains the name of the file to decompress. The file-level decoder is called, which is responsible for reading the file header information and decompressing the data to a buffer. The decompressed data form the bitplanes which need to be merged to obtain the symbols. If Gray coding was used in the encoder, then the symbol is transformed back to the binary representation. The inverse of bitplane reduction maps the symbols back to the original symbol. The result is the raw image data which should be identical to the original data. For confirmation, a comparison is done between the reconstructed data and the original data.

4.2.2.1.1 Bitplane merging

The decompressed data is written into an array by the file-level encoder after the decompressing process. The bitplanes need to be merged to obtain symbols of p bits per pixel. Figure 4.12 shows the pseudocode. Conversion from reflected Gray codes to binary representation is done if required. It is integrated into the same process.

The array `buffer[]` contains the decompressed data in bitplanes. The variable `use_gc` indicates whether Gray coding was applied.

Each bit of a symbol is read from the corresponding bitplane at the same pixel location. The bits are placed into their positions in the symbol using shift operation and bitwise OR. Conversion from reflected Gray code to binary code is also performed if required. The merged symbol is written into the array `data[]`.

```
/*bitplane merging*/
if (use Gray code)
   use_gc ← $1_2$
else
   use_gc ← $0_2$

for (i=0 to (HxW)-1){
  symbol ← 0
  for (p=0 to bpp-1)
    symbol←(symbol<<1)OR(buffer[p*H*W+i]XOR(use_gc AND symbol))
    data[i] ← symbol
}

/*inverse of bitplane reduction*/
for (i=0 to (HxW)-1)
   write new_orig[data[i]] to output file
```

Figure 4.12: Pseudocode for bitplane merging and inverse of bitplane reduction

4.2.2.1.2 Inverse of bitplane reduction

The inverse of bitplane reduction is performed to map the data back to the original symbol. This is done simply by using the translation table and performed on-the-fly when writing the symbol to the output file, as shown in Figure 4.12.

4.2.2.1.3 File comparison

To confirm the reconstructed data is identical to the original data, as should be the case in a lossless compression scheme, both files are compared numerically byte-by-byte. Figure 4.13 shows the pseudocode. If one of the bytes is not identical, then the files are not identical and a flag is set. Of course, both files should also be of the same size.

```
flag ← 0
if (decoded file size = original file size)
   while (not reached end-of-file)&(flag=0){
         A ← byte read from original file
         B ← byte read from decoded file
         if (A ≠ B)
           flag ←1
   }
else
   flag ←1
```

Figure 4.13: Pseudocode for file comparison

4.2.2.2 File-level decoder

The file-level decoder is responsible for managing the decoding the file as a whole. The flow of the function is given in Figure 4.14. Before starting the decompression, information regarding the file is read from the header. The functions `startinputtingbits( )` and `start_decode( )` are called for initialization purposes.

Then, the models which contain the probability tables are created and initialized. The bitstream is decompressed by calling the symbol-level decoder.

After the bitstream has been completely decoded, termination should be done using the functions `finish_decode()` and `doneinputtingbits()`. However, in Moffat *et al.*'s implementation, `finish_decode()` is an empty function that does nothing. The second function, `doneinputtingbits()`, is not necessary when there is only a single bitstream in the file. Thus, the functions are not used, saving two function calls.

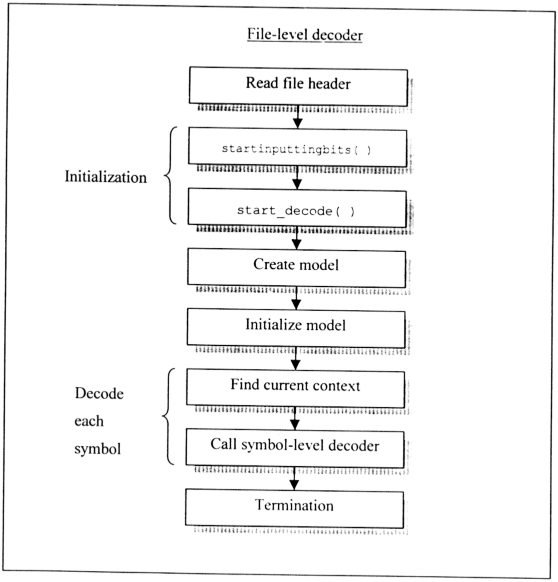


Figure 4.14: File-level decoder for bitplane coding

The decoder program has to use a model that is identical to that used by the encoder program in order to decode the data correctly. Therefore, the creation, initialization and management of the model are the same as described in Section 4.2.1.2.3.

4.2.2.2.1 Initialization

An input buffer is used to store the byte data read from the file to be decompressed. A pointer is used to point to the current bit of the buffer that is to be decoded, since the decoder treats the input data as a bitstream. When the pointer reaches the final bit, a new byte is read into the buffer from the file. The function `startinputtingbits()` simply initializes the pointer to the first bit to be decoded.

The function `start_decode( )` initializes the arithmetic decoder's range variables as required by the arithmetic decoder. The initial offset value, D, is initialized to 0 while the range, R, is initialized to 2^{b-1}. D is then filled with as many bits, via the input buffer, depending on how many bits are used to represent the coder's limits (see Section 3.3.1).

4.2.2.2.2 Decoding each symbol

The compressed bitstream is decoded by calling the symbol-level decoder. The successfully decoded symbols will be filled into an array, `buffer[]` which represents the *p* bitplanes. The bits are written into their appropriate bitplanes, starting from the MSB bitplane down to the LSB bitplane, as shown in the pseudocode in Figure 4.15.

As in the encoder, the probability tables are initialized before decoding begins for each new bitplane. The context is found from a neighbourhood template, similar to the encoder. The decoder uses the same neighbourhood template as the encoder.

```
ptr ← 0
for (p=0 to bpp-1){
  initialize model
  for (row=0 to H-1)
        for (column=0 to W-1){
              find the current context
              buffer[ptr] ←  symbol-level decoder( )
              ptr ← ptr+1
        }
}
```

Figure 4.15: Pseudocode for decoding each symbol

4.2.2.3 Symbol-level decoder

The symbol-level decoder is responsible for decoding a symbol by calling the arithmetic decoder and managing the model, as shown in Figure 4.16. First the function call `binary_arithmetic_decode( )` is made, which returns the decoded symbol. Then the probability table for the current context is updated and scaled down if necessary, in the same way as its counterpart in the compression program.

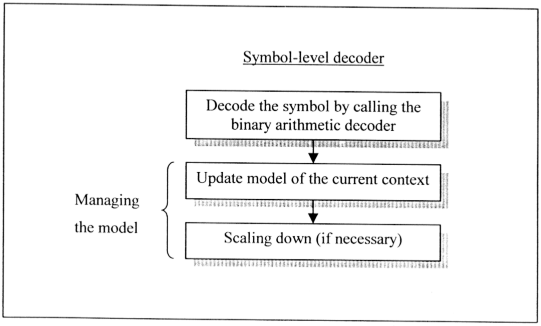

Symbol-level decoder
Decode the symbol by calling the binary arithmetic decoder
Update model of the current context
Scaling down (if necessary)
Managing the model

Figure 4.16: Symbol-level decoder for bitplane coding

4.3 Prediction by Partial Matching (PPM)

This section describes the implementation of the PPM compression and decompression program. Certain functions are identical to those described in previous sections. In this case, their description will not be repeated and the reader is kindly requested to refer to the relevant section.

4.3.1 Compression program

The implementation of the front end, file-level encoder and symbol level-encoder is described next.

4.3.1.1 Front end

The flow for the front end of the PPM compression program is given in Figure 4.17.

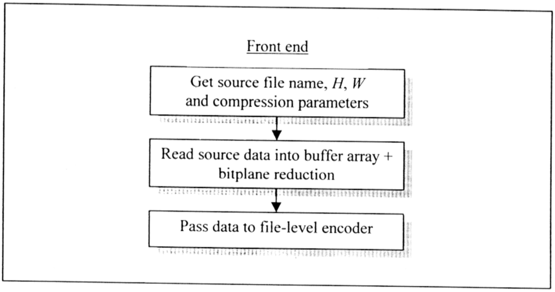

Front end
Get source file name, H, W and compression parameters
Read source data into buffer array + bitplane reduction
Pass data to file-level encoder

Figure 4.17: Front end of compression program for PPM

First, the program obtains from the user the name of the file to compress and the image's height, H and width, W. For experimenting with the compression parameters, the program should allow the parameters to be changed. For PPM, the parameters are:

i. escape method type (A, B, C or D)

ii. maximum order value, k

iii. neighbourhood template type

iv. update exclusion – full counting or single counting

v. frequency count scaling value

For item (i), (iii) and (iv), different versions of the program were compiled. Item (ii) and (v) were made as run-time options.

Data from the file is read into a buffer array and bitplane reduction is performed on the data, as in bitplane coding. For PPM, bitplane reduction is performed in order to bound the symbols' values to the range of $[0, s)$, where s is the number of unique colours. Thus, from the PPM compressor's point of view, the alphabet of the message to be compressed is comprised of the letters from the set $\{0, 1,\ldots, s\text{-}1\}$. The processed data is then passed to the file-level encoder.

4.3.1.2 File-level encoder

The file-level encoder is responsible for managing the encoding of the file as a whole. The flow of the function is the same as that of the bitplane coding given in Figure 4.6. Initializing and terminating the arithmetic coder use the same functions. The differences are in the implementation of the model, finding the current context and how each symbol is encoded.

4.3.1.2.1 Creation and initialization of the models

PPM uses a more complicated model compared to bitplane coding. As before, three things need to be done with regards to the models:

1. Creating the models
2. Initializing the models
3. Managing the models

Creating and initializing the model is common to the escape methods A, B, C and D. However, management of the model is slightly different for method D. Model management is done while encoding a symbol and will be described in Section 4.3.1.2.3.

Creating the models

Each model is a probability table containing statistics of the data being encoded. There is a model associated with each possible context for each order, from order -1, order 0, and so on, up to the maximum order *k*. The maximum order *k* is a predetermined value specified by the compression parameter.

This probability table implementation is based on the one used by Witten *et al.* (1987) and is adapted for use here. A C *struct* type is used to store the probability table, as shown by the C code snippet below:

```
typedef struct{
      unsigned short *freq;
      unsigned short *cum_freq;
      unsigned char *char_to_index;
      unsigned char *index_to_char;
} ppm_prob_table;
```

The structure is made up of four arrays. The function of each array is as follows:

1. `freq[ ]` – used to store the frequency count of each symbol
2. `cum_freq[ ]` – used to store the cumulative frequency count of the symbols.
3. `char_to_index[ ]` – translates a symbol to its index in the probability table.
4. `index_to_char[ ]` – translates the probability table's index to the associated symbol.

Only pointers to the array are declared so that the array size can be dynamically allocated during run-time. This is because it is only during run-time that we know how many symbols are used. The size dynamically allocated to `freq[]`, `cum_freq[ ]` and `index_to_char[ ]` is $s+1$, while for `char_to_index[ ]` the size is s.

The reason that a translation table is set up to translate between the symbol and its index in the probability table is that the probability table stores the statistics as a *move-to-front list*, ordered according to the symbols' frequency counts. The symbol with the largest frequency count occupies the 'top' of the table (given by `freq[1]`) followed by the symbol with next largest frequency count and so on. This way, according to Witten *et al.* (1987), the frequent symbols can be decoded with smaller number of execution loops. The element `cum_freq[0]` is used to store the total frequency count.

The probability tables are stored in a two dimensional array, as shown in an example in Figure 4.18. The row represents the order and the column represents the possible contexts for that order. Order 0 has only one probability table associated to it. For order 1 and higher, there is one probability table for each possible context. The number of possible contexts for an order n is given by s^n.

No probability table is kept for order-1. Instead the probabilities are found implicitly from the symbol to be encoded (as shall be described in Section 4.3.1.3.1).

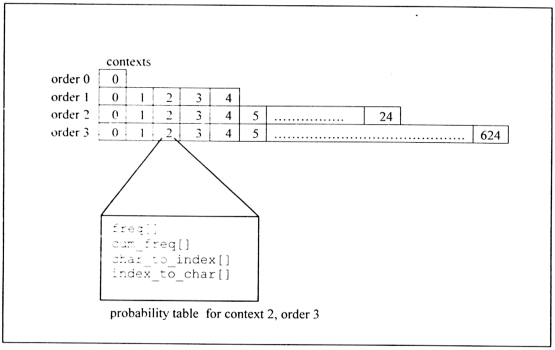


Figure 4.18: Example of how the probability tables are stored

Initializing the models

Initializing a model is shown by the pseudocode in Figure 4.19. It is divided into:

1. Setting up the translation tables, `char_to_index[ ]` and `index_to_char[]`
2. Initializing the elements of the `freq[ ]` and `cum_freq[ ]` arrays.

```
 /* setup translation table */
for (i=0 to s-1){
    char_to_index[i] ← i+1
    index_to_char[i+1] ← i
}

/*initialize freq[] and cum_freq[] */
for (i = 0 to s){
    freq[i] ← 0
    cum_freq[i] ← 0
}
```

Figure 4.19: Pseudocode for initializing a model

The `char_to_index[]` array is set up such that the array index represents a symbol and the corresponding element is the index for that symbol in the probability table. It is the other way around for the `index_to_char[ ]`. This way, given a symbol, its index in the probability table can be looked up using the `char_to_index[ ]`, and vice versa. Note that when a symbol *x* is referred to as preceding *y*, this means that the position of symbol *x* in the probability table is below *y* i.e. `char_to_index[x] > char_to_index[y]`.

4.3.1.2.2 Finding the context

The context for a symbol is found using a neighbourhood template. The template consists of a set of pixels in the neighborhood of the current pixel that is being encoded.

The values of the pixels forming the template are filled into an array, `neighbor[]`, as shown by the pseudocode in Figure 4.20. The array is referred to when calculating the current context for a given order. If a portion of the template lies outside the image, the default value 0 is used for the pixels that lie outside the image boundary.

```
for (i=0 to k-1 ){
   if (pixel is outside image boundary)
       neighbour[i] ← 0
   else
       neighbour[i] ← data[x]
}
```

`Note: data[]` is the data buffer, *x* gives the position of the neighbour pixel.

Figure 4.20: Pseudocode for filling in the `neighbour[]` array

The *conditional if* is used to find the neighbourhood pixels because it is easy to modify, hence suitable for testing purpose when experimenting with various templates. In an actual implementation, if the template and maximum order is fixed, it would be more efficient to embed the process of filling `neighbour[]` into the encoding loop of without the need to test whether a neighbour pixel is outside the image boundary.

The context is calculated as an *s*-ary digit number, which then serves as an array index to the corresponding probability table for that context. The following equation is used to calculate the value of the context:

$$\text{context} = \sum_{i=0}^{k-1} n_i s^i$$

where n_i denotes `neighbour[i]`, s denotes the number of symbols used and k is the maximum order..

The equation gives the order k context, which is the first context that needs to be calculated when encoding a symbol, since the PPM algorithm starts by attempting to encode using the largest order. If we need to find the context for the next lowest order, k-1, all that needs to be done is to subtract the value of the most significant digit from the context for order k. The subsequent lower orders are found in a similar fashion.

4.3.1.2.3 Encoding each symbol

Each symbol is encoded by calling the symbol-level encoder. The encoding follows the PPM algorithm, as shown in the pseudocode given in Figure 4.21.

```
ptr ← 0
for (i=0 to H-1)
 for (j=0 to W-1){
   flag ← 0
   current_order ← k
   sym ← data[ptr]
   find the current context
   ptr ← ptr+1
   do{
      if (freq[sym] > ZERO){
        initialize exclude[]  to 0
        find exclusion symbols
        call the symbol-level encoder to encode sym
        update model of current context and order
        flag ← 1
      }
      else{
        if (cum_freq[0]≠0){
              initialize exclude[] to 0
              find exclusion symbols
              call the symbol-level encoder to encode escape
              update model of current context and order
        }
        else
              update model of current context and order

        current_order ← current_order-1
        find the context for the lower order
      }
  }while (current_order ≥ 0 AND flag ≠1)

  if (current_order= -1)
       call the symbol-level encoder to encode sym

  if full counting
     for (o =  current_order-1 to 0 )
        update model

 }
```

Note:
All references to `freq[]`, `cum_freq[]`, `char_to_index[]` and `index_to_char[]` are for the `ppm_prob_table` of the current context of the current order
For methods A, C, and D: *ZERO = 0*
For methods B: *ZERO = 1*

Figure 4.21: Pseudocode for encoding a symbol using PPM

The function of each variable in the pseudocode is as follows:

1. `data[]` - the array which holds the symbols to be encoded
2. `ptr` - used to point to the current symbol being encoded.
3. `flag` - keeps track of whether a symbol has been successfully encoded
4. `current_order` - keeps track of the current order
5. `sym` - the symbol to be encoded
6. `exclude[]` - an array to mark the symbols for exclusion, to be used by the symbol-level encoder for probability calculation

The procedure for encoding a symbol is as follows. First, attempt to encode the symbol is made using order *k*, the maximum order. The symbol's frequency count is looked up in the probability table corresponding to the current context and order. If the count is such that the symbol is considered to have been 'seen' (i.e. a non-novel symbol) before in the current context, then it can be encoded. Otherwise, an escape symbol is encoded if required and encoding is attempted at the next lower order. This is repeated until the symbol is successfully encoded or order 0 is reached. If even with using order 0 the symbol is still not successfully encoded, then the symbol is encoded using order -1.

Note that a symbol is considered to have been 'seen' in the current context if its frequency count in the corresponding probability table is larger than 0 for method A, C and D, and larger than 1 for method B.

To encode the symbol, first the `exclude[]`array needs to be filled to mark the symbols for exclusion, so that the symbol-level encoder can apply the exclusion principle. This is done by finding the symbols that have already been 'seen' (i.e. non-novel symbols) in the previous higher order. Then, the symbol-level encoder is called.

Let order *n* be the order in which the symbol was successfully encoded, where $n \geq 0$. For each order *k* down to order *n*, the relevant probability tables corresponding to the current context will be updated. If full counting is applied, then for all the orders lower than order *n*, if any, the relevant probability tables corresponding to the current context will also be updated. However, if single counting is applied, this step is omitted.

For example, if $k = 4$ and $n = 2$, then in the case of full counting, the probability tables for all the orders 4, 3, 2, 1 and 0 are updated. In the case of single counting, only the probability tables for orders 4, 3 and 2 are updated.

4.3.1.2.4 Managing the model

The model is updated each time a symbol is encoded, whether it is a normal symbol or escape symbol. Updating consists of:

1. Scaling down the frequency counts when required
2. Incrementing the frequency count of the encoded symbol and moving the symbols' position to keep the probability table ordered as a move-to-front list

Updating is common for method A, B, and C, but slightly different for method D. The updating procedure is shown in the pseudocode of Figure 4.22, and is adapted from the implementation by Witten *et al.* (1987).

Scaling down

Scaling down the frequency counts is performed when a specified maximum value for the total frequency count has been reached, as shown in Figure 4.22

```
do for the current context and order:

/* if required, do scaling */
if (cum_freq[0] ≥ maximum value){
   cum ← 0
   for (i=s to 0){
        halve freq[i]
        cum_freq[i] ← cum
        cum ← cum + freq[i]
   }
}

/* find symbol's  new position. */
for (i=symbol_index to freq[i]=freq[i-1])
   i ← i-1
if(i=0) i←1
if (i<symbol_index){
   temp1 ← index_to_char[i]
   temp2← index_to_char[symbol_index]
   index_to_char[i] ← temp2
   index_to_char[symbol_index] ← temp1
   char_to_index[temp1] ← symbol_index
   char_to_index[temp2]← i
}

/*  increment the frequency count for the symbol */
increment freq[i] by increment_value
for (i=i-1 to 0)
   increment cum_freq[i] by increment_value
```

Note:

1. `halve freq[i]` is given by:
   Method A,B,C: `freq[i]←(freq[i]+1)>>1`
   Method D: `freq[i]←(freq[i]>>1) OR` 1_b

2. `symbol_index` is the symbol's index in the probability table

3. For Method A,B,C: `increment_value ← 1`
   For Method D: `if (freq[i]=0) increment_value ← 1`
   `else increment_value ← 2`

Figure 4.22: Pseudocode for updating a model

For method A, B, and C, scaling is done by halving the `freq[]` elements' values and taking the smallest integer that is equal or larger than the halved value (the ceiling function). For method D, it is done by halving the `freq[]` elements values, and taking the odd-numbered integer closest to the halved value. Additionally, `cum_freq[]` is updated to reflect the new frequency counts.

Incrementing the frequency count

Each time a symbol is encoded, its frequency count in the relevant probability table needs to be incremented by a predetermined amount. The amount depends on the escape method used, as shown in the pseudocode of Figure 4.22.

In order to keep the table ordered according to the frequency counts, the symbol's new position is found. If its new position is higher in the table compared to its present position, it will be moved, while `char_to_index[]` and `index_to_char[]` are updated accordingly. Its value in `freq[]` is then incremented and `cum_freq[]` is updated accordingly.

4.3.1.3 Symbol-level encoder

The symbol-level encoder is responsible for finding the symbol probability to be passed to the arithmetic encoder, given the symbol to be encoded and the probability table of the current context and the current order.

The function call to the arithmetic encoder is in the form of:

```
arithmetic_encode(l,h,t)
```

where `l`, `h` and `t` are the function parameters representing the symbol probability (as described in Section 3.3). The way the probability is found is handled differently for order -1 compared to order 0 and higher.

4.3.1.3.1 Order -1 encoder

The pseudocode for the order -1 encoder is shown in Figure 4.23. It encodes a symbol based on an equiprobable model, where each symbol is assigned the same frequency count of 1. Therefore, the total count `t` is equal to the number of symbols used *s*, so each symbol is encoded with the probability 1/*s*. Table 4.1 shows the probability table for the order -1 model.

```
j←0
k←0
for (i= 0 to s-1)
    if non-novel symbol
    {
        k ← k+1
        if(i<s)
          j ← j+1
    }
k ← k-1
arithmetic_encode(sym-j,sym+1-j,s-k)
```

Figure 4.23: Pseudocode for order -1 encoder

Table 4.1: Probability table for order -1 model

symbol	freq	cum_freq
0	1	0
1	1	1
2	1	2
. . .	. . .	. . .
s-1	1	s-1
		s

No probability table is actually kept for the order -1 model. This is because `l`, `h` and `t` can be trivially obtained. If the symbol to be encoded is `sym`, then:

```
l = sym
h = sym + 1
t = s
```

To apply the exclusion principle, the number of non-novel symbols is subtracted accordingly. Whether a symbol is non-novel is determined from the order 0 probability table. A symbol is considered as non-novel if its frequency count is more than 0 for method A, C, and D, and more than 1 for method D. With exclusion applied, the equations above become:

```
l = sym - j
h = sym + 1 - j
t = s - k
```

where,

`j` = number of non-novel symbols preceding *sym* in the probability table

`k` = total number of non-novel symbols

4.3.1.3.2 Order 0 and above encoder

There are two cases: encode an escape symbol or a normal symbol. In both cases, there are differences between the way the different escape methods find `l`, `h` and `t`. Therefore, three set of pseudocodes are presented, each for method A, B and C (method D is common with C)

Encoding escape symbol

Figure 4.24 shows the pseudocode for encoding an escape symbol. No entry is stored for the escape symbol in the probability tables. The escape symbol is encoded by placing an imaginary entry for it at the bottom of the probability table. Therefore the values of `l`, `h` and `t` have to be found accordingly, depending on which escape method is used, as summarized in Table 4.2.

To apply the exclusion principle, the excluded symbols' counts have to be subtracted from l, h and t. To find the frequency counts to be subtracted, the probability table for the current order and context is traversed until the value of freq[i] is 0. If a symbol is marked for exclusion, as indicated by the array exclude[], its contribution to the frequency count will be subtracted.

```
Method A:

k←0
for (i=1 to s){
  if (freq[i-th symbol] = 0)
        break
  if (the i-th symbol was marked for exclusion)
         k ← k + freq[i]
}
if (cum_freq[0] ≠ k)
   arithmetic_encode(0,1,cum_freq[0]-k+1)

Method B:

k←0
for (i=1 to s){
  if (freq[i-th symbol]= 0)
       break
  if (the i-th symbol was marked for exclusion)
         k ← k + freq[i]-1
}
e ← i-1
if (cum_freq[0] ≠ k+e)
  arithmetic_encode(0,e,cum_freq[0]-k)

Method C, D:

k←0
for (i=1 to s){
  if  (freq[i-th symbol]=0)
        break
  if (the i-th symbol was marked for exclusion)
         k ← k + freq[i]
}
e ← i-1
if (cum_freq[0] ≠ k)
   arithmetic_encode(0,e,cum_freq[0]-k+e)
```

Figure 4.24: Pseudocode for encoding an escape symbol

Table 4.2: Finding l, h and t for escape symbol in method A, B, C and D

escape method	A	B	C , D
l	0	0	0
h	1	e	e
t	cum_freq[0]-k+1	cum_freq[0]-k	cum_freq[0]-k+e

Where,

e = total number of symbols seen in the current context

k = total frequency counts of excluded symbols

Encoding a normal symbol

Figure 4.25 shows the pseudocode for encoding a normal symbol. Finding the frequency counts to be subtracted due to exclusion is done in a similar fashion as in encoding an escape symbol. The symbol is encoded keeping in mind that the escape symbol is placed as an imaginary entry at the bottom of the probability table. Table 4.3 summarizes how the values of l, h and t are found for encoding a normal symbol.

Table 4.3: Finding l, h and t for a normal symbol in method A, B, C and D

method:	A	B	C, D
l	cum_freq[sym]-j+1	cum_freq[sym]-j+sym	cum_freq[sym]-j+e
h	cum_freq[sym-1]-j+1	cum_freq[sym-1]-j+sym-1	cum_freq[sym-1]-j+e
t	cum_freq[0]-k+1	cum_freq[0]-k	cum_freq[0]-k+e

Where,

e = total number of symbols seen in the current context

j = total frequency counts of excluded symbols preceding *sym* in the probability table

k = total frequency counts of excluded symbols

Method A:

```
j←0
k←0
for (i=1 to s ){
  if (freq[i-th symbol] = 0)
    break
  if (the i-th symbol was marked for exclusion){
    k ← k + freq[i]
    if (i>sym)  j← j + freq[i]
  }
}
arithmetic_encode(cum_freq[sym]-j+1,cum_freq[sym-1]-j+1,
   cum_freq[0]-k+1)
```

Method B:

```
j←0
k←0
for (i=1 to s ){
  if  (freq[i-th symbol] = 0)
      break
  if (the i-th symbol was marked for exclusion)
      if (freq[i]>1){
         k ← k+ freq[i]-1
         if (i>sym) j← j + freq[i]-1
      }
}
arithmetic_encode(cum_freq[sym]-j+sym,cum_freq[sym-1]-j+sym-1,
   cum_freq[0]-k)
```

Method C, D:

```
j←0
k←0
for (i=1 to s ){
  if (freq[i-th symbol] = 0)
      break
  if (the i-th symbol was marked for exclusion){
      k ← k+ freq[i]
      if (i>sym) j← j + freq[i]
  }
}
e ← i-1
arithmetic_encode(cum_freq[sym]-j+e,cum_freq[sym-1]-j+e,
   cum_freq[0]-k+e)
```

Figure 4.25: Pseudocode for encoding a normal symbol

4.3.2 Decompression program

Implementation of the decompression program for PPM is described next.

4.3.2.1 Front end

The flow of the front end for the PPM decoder is given in Figure 4.26. The blocks have the same functions as that of the front end in bitplane coding, except bitplane merging is not included.

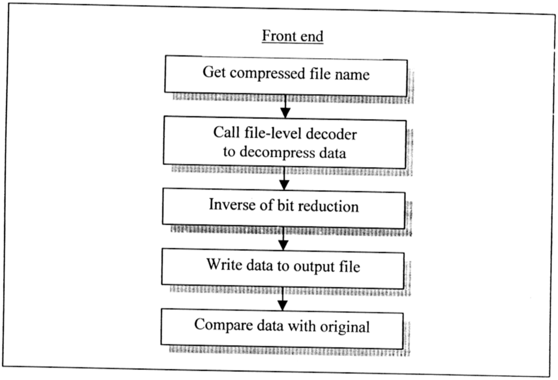

Front end
Get compressed file name
Call file-level decoder to decompress data
Inverse of bit reduction
Write data to output file
Compare data with original

Figure 4.26: Front end of the decompression program for PPM

4.3.2.2 File-level decoder

The file-level decoder is responsible for managing the decoding of the file as a whole. The flow of the function is same as that of the bitplane coding given in Figure 4.14. The only blocks which are different are the implementation of the model, finding the current context and how each symbol is decoded. Of these, the first two is the same as described for the compression program in Section 4.3.1.2. Therefore, only the decoding of each symbol using PPM is described next.

4.3.2.2.1 Decoding each symbol

Each symbol is decoded by calling the symbol-level decoder. The decoding follows the PPM algorithm, as shown in the pseudocode given in Figure 4.27.

The functions of the variables in the pseudocode are:

1. `data[]` - the array which holds the symbols that have been decoded
2. `ptr` - used to point to the current symbol to be decoded.
3. `flag` - keep track of whether a symbol has been successfully decoded
4. `current_order` - keep track of the current order
5. `sym` - the symbol to be decoded
6. `exclude[]` - an array to mark the symbols for exclusion, to be used by the symbol level decoder for probability calculation

Decoding is performed as follows. First, an attempt is made to decode the symbol using order k and the corresponding context. If an escape symbol is obtained, the next lower order is tried. This is repeated until the symbol is successfully decoded or order 0 is reached. If even with using order 0 the symbol is still not successfully decoded, then the symbol is decoded using order -1.

The probability tables are updated in lockstep with the encoder. Calculating the current context is same as that of the encoder side, making use of the same neighbourhood template that the encoder employed.

```
ptr ← 0
for (i=0 to H-1)
 for (j=0 to W-1){
  flag ← 0
  current_order ← k
  find the current context
  do{
     if (cum_freq[0]=0){
        current_order ← current_order-1
        find the lower order context
     }
     else{
        initialize exclude[]to 0
        find exclusion symbols
        sym ← symbol_level_decoder( )
        if (sym is not escape) flag ←1
        else{
          current_order ← current_order-1
          find the lower order context
        }
     }
  }while (current_order≥0 AND flag≠1)

  if (current_order= -1){
     sym ← symbol_level_decoder( )
     current_order ← 0
  }

  data[ptr] ← sym
  ptr ← ptr+1

  for (o=k to current_order )
     update model

  if  full counting
     for (o=current_order-1 to 0)
        update model

 }
```

Figure 4.27: Decoding each symbol in PPM

4.3.2.3 Symbol-level decoder

The symbol-level decoder is responsible for finding the parameters to be passed to the arithmetic decoder. The arithmetic decoder does not actually return the decoded symbol, but rather a target value which points to where the symbol is in the probability table. Therefore the symbol-level decoder will also need to find what the decoded symbol actually is.

The arithmetic decoder has two functions that need to be called:

```
arithmetic_decode_target(t)
arithmetic_decode(l,h,t)
```

where `l`, `h` and `t` are the function parameters representing the symbol probability (as described in Section 3.3). The first function returns the target value, while the second updates the arithmetic coder's state. The way the probability is found and the symbol is decoded is different for order -1 compared to order 0 and higher.

4.3.2.3.1 Order -1 decoder

Similar to its encoder counterpart, the order -1 decoder decodes a symbol based on an equiprobable model, where each symbol is assigned the frequency count of 1 and no probability table is actually kept for the order -1 model. The pseudocode is shown in Figure 4.28.

```
j←0
k←0
for (i=0 to s-1)
   if non-novel symbol
      k ← k+1
target ← arithmetic_decode_target(s-k)
arithmetic_decode(target,target+1,s-k)

for (j=0 to target)
   if non-novel symbol
      target ← target+1
return target
```

Figure 4.28: Pseudocode for order -1 decoder

First, the frequency count contributed by the excluded symbols, *k*, is found. The function `arithmetic_decode_target(t)` is called, which returns an integer value that lies in the range [l,h) that was used at the corresponding call to `arithmetic_encode()`. Then, `arithmetic_decode(l,h,t)` is called, to adjust the decoder's state variables to reflect the changes made in the encoder during the corresponding call to `arithmetic_encode()`.

For the order -1 model, `l`, `h` and `t` can be trivially obtained from the decoded symbol `sym`. To account for the exclusion principle, the number of non-novel symbols will have to be subtracted accordingly. Whether a symbol is non-novel is determined from the order 0 probability table. A symbol is considered as non-novel if its frequency count is more than 0 for method A, C, and D, and more than 1 for method B. With exclusion applied, the equation for `l`, `h` and `t` are:

```
        l = sym - j
        h = sym +1 - j
        t = s - k
```

where,

`j` = number of non-novel symbols preceding `sym` in the probability table

`k` = total number of non- novel symbols

The arithmetic decoder returns the value `target = sym-j`. From this and the knowledge of which symbols were excluded, the actual symbol `sym` can be found by adding the value of `j` to `target`.

4.3.2.3.2 Order 0 and above decoder

Three set of pseudocodes are presented, each for escape method A, B and C (method D is common with C), as shown in Figure 4.29, 4.30 and 4.31 respectively. Since no entry is stored for the escape symbol in the probability tables, the values of [illegible] and [illegible] have to be found accordingly, as summarized in Table 4.2 and 4.3 respectively, same as the encoder.

```
Method A:
j←0
k←0
for (i=1 to s ){
  if (freq[i] =0)
        break
  if (the i-th symbol was marked for exclusion
        k = k+ freq[i]
}
if (cum_freq[0]= k)
  return escape

target← arithmetic_decode_target(cum_freq[0]-k+1
if (target<1){
  arithmetic_decode(0,1,cum_freq[0]-k+1)
  return escape
}
else{
  for (sym=1 to cum_freq[sym]>target-1)
          sym ← sym+1
  for (i=s to sym)
          if i-th symbol was marked for exclusion
                  j← j+ freq[i]
                  for (sym = 1 to cum_freq[sym]>target-j-1)
                          sym ← sym+1
          }
  arithmetic_decode(cum_freq[sym]-j+1,
          cum_freq[sym-1]-j+1, cum_freq[0]-k+1)
  return sym
}
```

Figure 4.29: Pseudocode for order 0 and above decoder in method A

First the total number of non-novel symbols, k, is found based on the exclusion array. If the value k equals to the total count t, then an escape symbol is generated, without calling the arithmetic decoder, mimicking the action taken by the encoder. Otherwise, arithmetic_decode_target(t) is called, which returns an integer target that lies in the range [l,h) that was used at the corresponding call to arithmetic_encode( ).

From the target value, a decision is made whether the decoded symbol is an escape symbol or a normal symbol. If it is a normal symbol, then what the symbol actually is needs to be found, based on the knowledge of which symbols were excluded from the frequency count by the encoder. In any case, arithmetic_decode(l,h,t) is called to adjust the decoder's state variables to reflect the changes made in the encoder during the corresponding call to arithmetic_encode( ).

```
Method B:

j←0
k←0
for (i=1 to s ){
  if (freq[i]=0)
      break
  if (the i-th symbol was marked for exclusion)
     if (freq[i]>1)
         k = k + freq[i]-1
}
e ← i-1
if (cum_freq[0]  = k+e)
   return escape

target ← arithmetic_decode_target(cum_freq[0]-k)
if (target < e){
    arithmetic_decode(0,e, cum_freq[0]-k)
    return escape
}
else{
   for (sym = 1 to cum_freq[sym]>target-sym )
        sym ← sym+1
   for (i=s to sym)
        if i-th symbol was marked for exclusion{
              j← j+ freq[i]-1
              for(sym=1 to cum_freq[sym]>target+j-sym)
                   sym ← sym+1
        }
    arithmetic_decode(cum_freq[sym]-j+sym,
        cum_freq[sym-1]-j+sym-1, cum_freq[0]-k)
   return sym
}
```

Figure 4.30: Pseudocode for order 0 and above decoder in method B

```
Method C, D:

j←0
k←0
for (i=1 to s ){
  if (freq[i]=0)
       break
  if (the i-th symbol was marked for [illegible]
       k = k + freq[i]
}
e ←i-1
if (cum_freq[0]=k)
    return escape
target = arithmetic_decode_target [illegible]
if (target<e) {
    arithmetic_decode(0,e, cum_freq [illegible]
    return escape
}
else{
   for (sym = 1 to cum_freq[s] [illegible]
        sym ← sym+1
   for (i=s to sym){
        if (the i-th symbol [illegible]sion)
              j= j+ freq[i]
        for (sym = 1 to cum_freq[s] [illegible]target-e-j)
              sym ← sym+1
   }
   arithmetic_decode(cum_freq[sym] [illegible]e,
        cum_freq[sym-1]-j+e, cum_freq[0]-k+e
   return sym
}
```

Figure 4.31: Pseudocode for order 0 and above decoder in method C and D

4.4 Combined method of bitplane coding and PPM

This section describes the implementation that combines both bitplane coding and PPM in a single compressor/decompressor. Since many functions are identical to those described in previous sections, their description will not be repeated here. The main differences are in the plane splitting/merging process and the file-level encoder/decoder.

4.4.1 Compression program

The front end and file-level encoder are described next. The symbol-level encoder follows the respective methods, which have been explained in earlier sections.

4.4.1.1 Front end

The flow for the front end of the compression program for the combined method is given in Figure 4.32. The steps are similar to those explained in the previous sections, except for the plane splitting process.

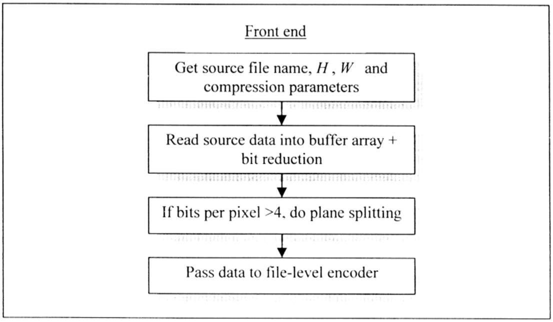

Front end
Get source file name, H, W and compression parameters
Read source data into buffer array + bit reduction
If bits per pixel >4, do plane splitting
Pass data to file-level encoder

Figure 4.32: Front end of compression program for the combined method

The plane splitting process for this method separates the bitplanes in two separate parts as mentioned in Section 3.8. Figure 4.33 shows the pseudocode for the process.

The variable bitplanes gives the number of bitplanes for bitplane coding and is equal to the bits per pixel (as found from the bitplane reduction process) minus 4. The bitplanes are then extracted, followed by the plane for PPM. Note that if the bits per pixel are equal to or less than 4 (i.e the number of symbol, s $\leq$ 16), the plane splitting process will not be performed.

```
bitplanes ← bpp-4
for (i=0 to (HxW -1
  temp ← buffer[i
  bit_number← bpp
  for (p=0 to bitplanes-1){
     bit_number ← bit_number-1
     dest[p*H*W+i]=(temp>>bit_number)AND $1_2$
   }
  dest[p*H*W+i]= temp AND $1111_2$
}
```

Figure 4.33: Pseudocode for plane splitting process

4.4.1.2 File-level encoder

The file-level encoder is responsible for managing the encoding of the file as a whole. Two encoders, the bitplane encoder and PPM encoder, are combined. Bitplane coding is performed, if required, followed by PPM, both operating independently, as shown in Figure 4.34. Each of the blocks for bitplane coding and PPM is identical to the steps shown in Figure 4.6, but with the header written once only. Bitplane coding is performed on p-4 bitplanes, where p is the bits per pixel and $p > 4$. If $p \leq 4$, bitplane coding is skipped and only PPM is performed.

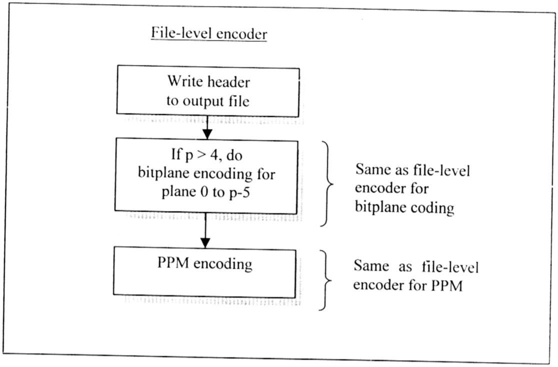

File-level encoder
Write header to output file
If p > 4, do bitplane encoding for plane 0 to p-5
Same as file-level encoder for bitplane coding
PPM encoding
Same as file-level encoder for PPM

Figure 4.34: File-level encoder for the combined method

4.4.2 Decompression program

As in the compression program, only the front end and file-level encoder is described. The symbol-level decoder follows the respective methods, which have been explained in earlier sections.

4.4.2.1 Front end

The program flow for the front end of the decompression program for the combined method is given in Figure 4.35. The functions are similar to those explained in the previous sections, except for the plane merging process.

The plane merging process merges the planes to obtain the symbol of *p* bits per pixel. Figure 4.36 shows the pseudocode for the process. Each bit of the symbol is read from the corresponding plane at the same pixel location. The bits are placed into their positions in the symbol using shift operation and bitwise OR. Finally, the merged symbol is written into the array `data[]`.

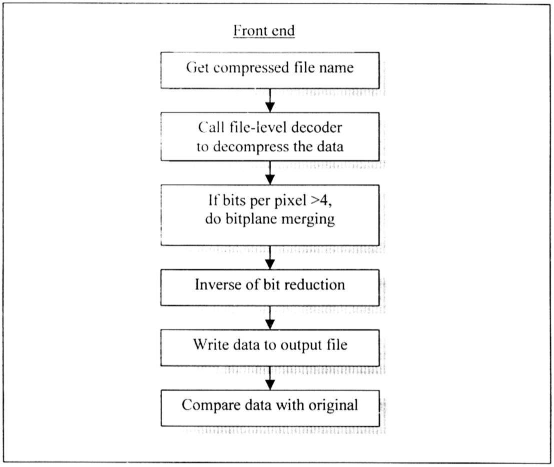

Front end
Get compressed file name
Call file-level decoder to decompress the data
If bits per pixel >4, do bitplane merging
Inverse of bit reduction
Write data to output file
Compare data with original

Figure 4.35: Front end of decompression program for the combined method

```
bitplanes = bpp-4
for [illegible] HxW)-1){
  temp = [illegible]
  for (p=[illegible] to bitplanes-1)
        temp = (temp << 1)OR(buffer[p*H*W+i])
   data[i] = (temp<<4)OR(buffer[p*H*W+i]AND 1111$_2$)
}
```

Figure 4.36: Pseudocode for plane merging process

4.4.2.2 File-level decoder

The file-level decoder is responsible for managing the decoding of the file as a whole. Since two separate encoders may have been used during the compression process, there are possibly two separate bitstreams that need to be decoded. Mirroring the encoder side, bitplane decoding is performed first, if required, followed by PPM, as shown in Figure 4.37.

Each of the blocks for bitplane coding and PPM is identical to the steps shown in Figure 4.14. Of course, the file header information is read only once. If $p>4$, the first bitstream is decoded using bitplane coding, followed by PPM for the second bitstream. If $p \leq 4$, it means that there is only one bitstream, so only PPM decoding is performed.

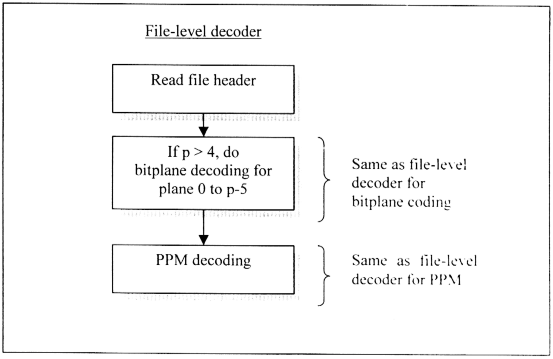

File-level decoder
Read file header
If p > 4, do bitplane decoding for plane 0 to p-5
Same as file-level decoder for bitplane coding
PPM decoding
Same as file-level decoder for PPM

Figure 4.37: File-level decoder for the combined method