6 Huffman/Lempel-Ziv Compression Methods

6.1 Introduction

Normally, general data compression does not take into account the type of data which is being compressed and is lossless. It can be applied to computer data files, documents, images, and so on. The two main techniques are statistical coding and repetitive sequence suppression. This chapter discusses two of the most widely used methods for general data compression: Huffman coding and Lempel-Ziv coding.

6.2 Huffman coding

Huffman coding uses a variable length code for each of the elements within the data. This normally involves analyzing the data to determine the probability of its elements. The most probable elements are coded with a few bits and the least probable coded with a greater number of bits. This could be done on a character-by-character basis, in a text file, or could be achieved on a byte-by-byte basis for other files.

The following example relates to characters. First, the textual data is scanned to determine the number of occurrences of a given letter. For example:

Letter:	'b'	'c'	'e'	'i'	'o'	'p'
No. of occurrences:	12	3	57	51	33	20

Next the characters are arranged in order of their number of occurrences, such as:

'e'	'i'	'o'	'p'	'b'	'c'
57	51	33	20	12	3

After this the two least probable characters are assigned either a 0 or a 1. Figure 6.1 shows that the least probable ('c') has been assigned a 0 and the next least probable ('b') has been assigned a 1. The addition of the number of occurrences for these is then taken into the next column and the occurrence values are again arranged in descending order (that is, 57, 51, 33, 20 and 15). As with the first column, the least probable occurrence is assigned a 0 and the next least probable occurrence is assigned a 1. This continues until the last column. When complete, the Huffman-coded values are read from left to right and the bits are listed from right to left.

The final coding will be:

'e'	11	'i'	10
'o'	00	'p'	011
'b'	0101	'c'	0100

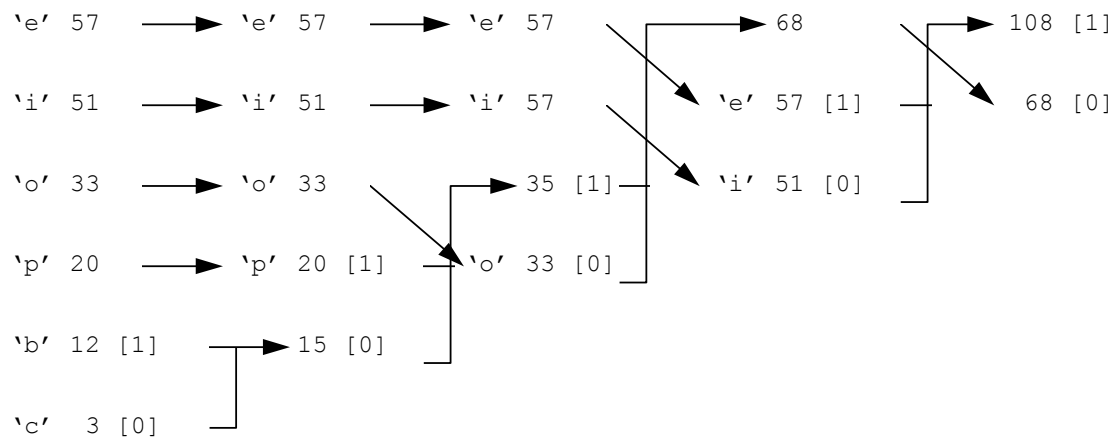


Figure 6.1 Huffman coding example

The great advantage of Huffman coding is that, although each character is coded with a different number of bits, the receiver will automatically determine the character whatever their order. For example, if a 1 is followed by a 1 then the received character is an 'e'. If it is then followed by two 0s then it is an 'o'. Here is an example:

```
11000110100100110100
```

will be decoded as:

```
'e'    'o'    'p'    'c'   'i'   'p'    'c'
```

When transmitting or storing Huffman-coded data, the coding table needs to be stored with the data (if the table is generated dynamically). It is generally a good compression technique but it does not take into account higher-order associations between characters. For example, the character 'q' is normally followed by the character 'u' (apart from words such as *Iraq*). An efficient coding scheme for text would be to encode a single character 'q' with a longer bit sequence than a 'qu' sequence.

In a previous example, we used soccer matches as an example of how data could be compressed. In a small sample of soccer matches the following resulted:

0 goals – 21 times; 1 goal – 34 times; 2 goals– 15 times; 3 goals– 14 times;
4 goals – 5 times; 5 goals – 2 times; 6 goals – 1 time.

We could then order them as follows:

1 goal – 34 times; 0 goals – 21 times; 2 goals –15 times; 3 goals – 14 times;
4 goals – 5 times; 5 goals – 2 times; 6 goals – 1 time.

This is obviously a small sample, and there are thus no codes for seven goals or more. With a

larger sample, there would be an associated number of occurrences. Figure 6.2 shows the resulting Huffman coding for these results. Thus, for example, a binary value of 01 will represent zero goals scored, and so on. This code could be combined with a table of values that represent each of the soccer teams. So, if we have 256 different teams (from 0 to 255), and use 00000000b to represent Mulchester, and 00000001b to represent Madchester, then the result:

Mulchester 2 Madchester 0

would be coded as:

00000000 **00** 00000001 **01**

where the bold digits represent the score. If the next match between the two teams resulted in a score of 4–1 then the code would be:

00000000 **1001** 00000001 **11**

Notice that the number of bits used to code the score can vary in size, but as we use a Huffman code, we can automatically detect the number of bits that it has. A problem with Huffman, is when we lose synchronization between the encoder and the decoder, thus, there must be occasional re-synchronization between them. One method that is used in HDLC is to use a special start and end bit sequence of 0111111, which cannot occur at any other place in the data.

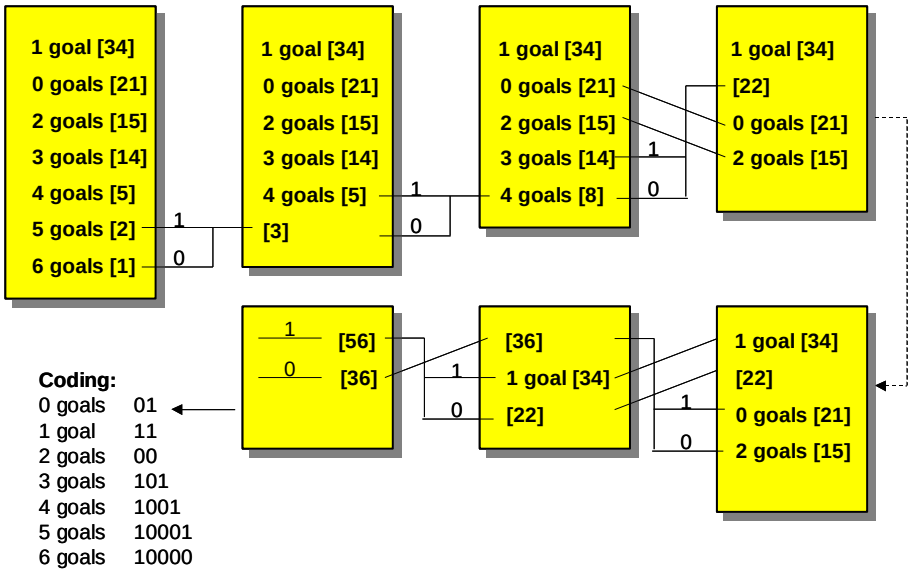


Figure 6.2 Huffman coding example

6.3 Adaptive Huffman coding

Adaptive Huffman coding was first conceived by Faller and Gallager and then further refined by Knuth (so it is often called the FGK algorithm). It uses defined word schemes which determine the mapping from source messages to code words. These mappings are based upon a running estimate of the source message probabilities. The code is adaptive and changes to remain optimal for the current estimates. In this way, the adaptive Huffman codes respond to locality and the encoder thus learns the characteristics of the source data. It is thus important that the decoder learns the encoding along with the encoder. This will be achieved by continually updating the Huffman tree to stay in synchronization with the encoder.

A second advantage of adaptive Huffman coding is that it only requires a single pass over the data. In many cases, the adaptive Huffman method actually gives a better performance, in terms of number of bits transmitted, than static Huffman coding.

6.4 Lempel-Ziv coding

Around 1977, Abraham Lempel and Jacob Ziv developed the Lempel–Ziv class of adaptive dictionary data compression techniques (also known as LZ-77 coding), which are now some of the most popular compression techniques. The LZ coding scheme is especially suited to data which has a high degree of repetition, and makes back references to these repeated parts. Typically a flag is normally used to identify coded and unencoded parts, where the flag creates back references to the repeated sequence. An example piece of text could be:

‘The **recei**ver **re**qui**re**s a **recei**pt for **it**. This **is** automatically s**en**t wh**en it is recei**ved.’

This text has several repeated sequences, such as ‘ is ’, ‘it’, ‘en’, ‘re’ and ‘ receiv’. For example, the repetitive sequence *recei* (as shown by the underlined highlight), and the encoded sequence could be modified with the flag sequence #*m*#*n* where *m* represents the number of characters to trace back to find the character sequence and *n* the number of replaced characters. Thus, the encoded message could become:

‘The receiver**#9#3**quires a**#20#5**pt for it. This is automatically sent wh**#6#2** it **#30#2#47#5**ved.’

Normally, a long sequence of text has many repeated words and phrases, such as ‘and’, ‘there’, and so on. Note that in some cases, this could lead to longer files if short sequences were replaced with codes that were longer than the actual sequence itself.

Using the previous example of sport results:

Mulchester 3 Madchester 2
Smellmore Wanderers 60 Drinksome Wanderers 23

we could compress this with:

Mulchester 3 Mad#13#7 2
Smellmore Wanderers 60 Drinksome#23#1123

6.5 Lempel–Ziv–Welsh coding

The Lempel–Ziv–Welsh (LZW) algorithm (also known LZ-78) builds a dictionary of frequently used groups of characters (or 8-bit binary values). Before the file is decoded, the compression dictionary is sent (if transmitting data) or stored (if data is being stored). This method is good at compressing text files because text files contain ASCII characters (which are stored as 8-bit binary values) but not so good for graphics files, which may have repeating patterns of binary digits that might not be multiples of 8 bits.

A simple example is to use a six-character alphabet and a 16-entry dictionary, thus the resulting code word will have 4 bits. If the transmitted message is:

```
ababacdcdaaaaaaef
```

Then, the transmitter and receiver would initially add the following to its dictionary:

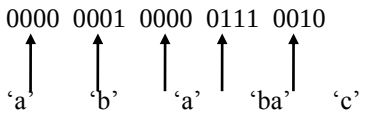
0000	‘a’	0001	‘b’
0010	‘c’	0011	‘d’
0100	‘e’	0101	‘f’
0110–1111	empty		

First the ‘a’ character is sent with 0000, next the ‘b’ character is sent and the transmitter checks to see that the ‘ab’ sequence has been stored in the dictionary. As it has not, it adds ‘ab’ to the dictionary, to give:

0000	‘a’	0001	‘b’
0010	‘c’	0011	‘d’
0100	‘e’	0101	‘f’
0110	‘ab’	0111–1111	empty

The receiver will also add this to its table (thus, the transmitter and receiver will always have the same tables). Next, the transmitter reads the ‘a’ character and checks to see if the ‘ba’ sequence is in the code table. As it is not, it transmits the ‘a’ character as 0000, adds the ‘ba’ sequence to the dictionary, which will now contain:

0000	‘a’
0001	‘b’
0010	‘c’
0011	‘d’
0100	‘e’
0101	‘f’
0110	‘ab’
0111	‘ba’
1000–1111	empty



Next, the transmitter reads the ‘b’ character and checks to see if the ‘ba’ sequence is in the table. As it is, it will transmit the code table address which identifies it, i.e. 0111. When this is received, the receiver detects that it is in its dictionary and it knows that the addressed sequence is ‘ba’.

Next, the transmitter reads a ‘c’ and checks for the character in its dictionary. As it is included, it transmits its address, i.e. 0010. When this is received, the receiver checks its dictionary and locates the character ‘c’. This then continues with the transmitter and receiver maintaining identical copies of their dictionaries. A great deal of compression occurs when sending a sequence of one character, such as a long sequence of ‘a’.

Typically, in a practical implementation of LZW, the dictionary size for LZW starts at 4 K (4096). The dictionary then stores bytes from 0 to 255 and the addresses 256 to 4095 are used for strings (which can contain two or more characters). As there are 4096 entries then it is a 12-bit coding scheme (0 to 4096 gives 0 to 2^{12}–1 different addresses).

6.6 Variable-length-code LZW compression

The Variable-length-code LZW (VLC-LZW) uses a variation of the LZW algorithm where variable-length codes are used to replace patterns detected in the original data. It uses a dictionary constructed from the patterns encountered in the original data. Each new pattern is entered into it and its indexed address is used to replace it in the compressed stream. The transmitter and receiver maintain the same dictionary.

The VLC part of the algorithm is based on an initial code size (the LZW initial code size), which specifies the initial number of bits used for the compression codes. When the number of patterns detected by the compressor in the input stream exceeds the number of patterns encodable with the current number of bits then the number of bits per LZW code is increased by one. The code size is initially transmitted (or stored) so that the receiver (or uncompressor) knows the size of the dictionary and the length of the codewords.

In 1985, the LZW algorithm was patented by the Sperry Corp. It is used by the GIF file format and is similar to the technique used to compress data in V.42bis modems.

6.7 Disadvantages with LZ compression

LZ compression substitutes the detected repeated patterns with references to a dictionary. Unfortunately the larger the dictionary, the greater the number of bits that are necessary for

the references. The optimal size of the dictionary also varies for different types of data; the more variable the data, the smaller the optimal size of the directory.

6.8 Practical Lempel-Ziv/Huffman coding

This section contains practical examples of programs which use Lempel-Ziv and/or Huffman coding. Most compression programs use either one or both of these techniques. As previously mentioned, both techniques are lossless. In general, Huffman is the most efficient but requires two passes over the data, while Lempel-Ziv uses just one pass. This feature of a single pass is obviously important when saving to a hard disk drive or when encoding and decoding data in real-time communications. One of the most widely used variants is LZS, owned by Stac Electronics (who were the first commercial company to produce a compressed drive, named Stacker). Microsoft have included a variation of this program, called DoubleSpace in DOS Version 6 and DriveSpace in Windows 95.

The LZS technique is typically used in mass backup devices, such as tape drives, where the compression can either be implemented in hardware or in software. This typically allows the tape to store at least twice the quoted physical capacity of the tape.

The amount of compression, of course, depends on the type of file being compressed. Random data, such as executable programs or object code files, typically has low compression (resulting in a file which is 50 to 95% of the original file size). Still images and animation files tend to have high compression and typically result in a file which is between only 2 and 20% of the original file size. It should be noted that once a file has been compressed there is virtually no gain in compressing it again (unless a differential method is used). Thus storing or transmitting compressed files over a system which has further compression will not increase the compression ratio (unless another algorithm is used).

Typical files produced from LZ77/LZ78 compression methods are ZIP, ARJ, LZH, Z, and so on. Huffman is used in ARC and PKARC utilities, and in the UNIX compact command.

6.8.1 Lempel-Ziv/Huffman practical compression

DriveSpace and DoubleSpace are programs used in PC systems to compress files on hard disk drives. They use a mixture of Huffman and Lempel-Ziv coding, where Huffman codes are used to differentiate between data (literal values) and back references and LZ coding is used for back references.

A DriveSpace disk starts with a 4 byte magic number (52 B2 00 00 08h). This identifies that the disk is using DriveSpace. Following this there are either literal values or back references which are preceded by control bits of either `0`, `11`, `100`, `1010` or `1011` (these are Huffman values coded to differentiate them). If the control bit is 0 then the following 7 bits of `abcdefg` correspond to data of `0gfedcba`, else if it is 11 then the following 7 bits of `abcdefg` correspond to data `1gfedcba`. Thus the data:

```
10110101 01111110 11100000 11111111
```

would be encoded as:

```
11 1010110 0 0111111 11 0000011 11 1111111
```

The back-reference values are preceded by `100`, `1010` or `1011`. If preceded by `100` then followed by `abcdef`*X*, which is a 6-bit back reference of a length given by *X*. A `1010` followed by `abcdefgh`*X* is an 8-bit back reference of 64+`hgfedcba` with length given by *X*. A `1011` followed by `abcdefghijkl`*X* is a 12-bit back reference of 64+256+ `lkjihgfedcba` with length given by *X*.

The back reference consists of a code indicating the number of bits back to find the start of the referenced data, followed by the length of the data itself. This code consists of *N* zeros followed by a 1. The number of zeros, *N*, indicates the number of bits of length data and the length of the back reference is $M+2^N+2$, where *M* is the *N*-bit unsigned number comprising the data length. Thus the minimum length of a back reference will be when $M=0$ and $N=0$ giving a value of 3. An example format of a back pointer is:

```
100  abcdef 000001 ghijk
```

where *N* will be 5 since there are five zeros after the 5-bit back reference and `fedcba` corresponds to the back reference `fedcba`. The length of the reference values will be $M+2^5+2$, where *M* is the 5-digit unsigned binary number `kjihg`. For example if the stored bit field were:

```
0101001010110010000000000000000000010010000100110010000010
0001001100101000010
```

It would be decoded as:

0101001010110010000000000000000000001001
MAGIC NUMBER

0 0001001	**100** 100000 1
'H' (100 1000)	As the control bit field is **100** then it has a 6-bit back reference of `000001` (one place back) followed by `1` which shows that the back reference length of bits is 0. Thus, using the formula $M+2^N+2$ gives $0+2^0+2=3$. The back reference has a length of 3 bytes, giving the output 'HHH'

0 0001001	**100** 1010000 10
'E' (`01010001`)	As the control bit field is **100**, it has a 6-bit back reference of `000101` (five places back) followed by `01` which shows that the back-reference length of bits is `1`. Thus, using the formula $M+2^N+2$ gives $0+2^1+2=4$. The character five places back is an 'H', thus 'H' is repeated four times.

This then gives the sequence 'HHHHEHHHH'.

In DriveSpace, each of the fields after the magic number is a group. A group consist of a control part (the Huffman code) and an item. An item may be either a literal item or a copy item (i.e. a 6, 8 or 12 bit back reference). The end of a file in DriveSpace is identified with a special 12-bit back-reference value of 1111 1111 1111 1111 (FFFFh).

6.8.2 GIF files

The graphic interface format (GIF) uses a compression algorithm based on the Lempel-Ziv-Welsh (LZW) compression scheme. When compressing an image the compression program maintains a list of substrings that have been found previously. When a repeated string is found, the referred item is replaced with a pointer to the original. Since images tends to contain many repeated values, the GIF format is a good compression technique.

6.8.3 UNIX compress/uncompress

The UNIX programs `compress` and `uncompress` use adaptive Lempel-Ziv coding. They are generally better than `pack` and `unpack` which are based on Huffman coding. Where possible, the `compress` program adds a `.Z` onto a file when compressed. Compressed files can be restored using the `uncompress` or `zcat` programs.

6.8.4 UNIX archive/zoo

The UNIX-based `zoo` freeware file compression utility employs the Lempel-Ziv algorithm. It can store and selectively extract multiple generations of the same file. Data can thus be recovered from damaged archives by skipping the damaged portion and locating undamaged data (using the `fiz` program).