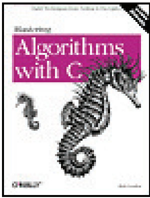


14.4 Description of Huffman Coding

One of the oldest and most elegant forms of data compression is Huffman coding, an algorithm based on minimum redundancy coding. Minimum redundancy coding suggests that if we know how often different symbols occur in a set of data, we can represent the symbols in a way that makes the data require less space. The idea is to encode symbols that occur more frequently with fewer bits than those that occur less frequently. It is important to realize that a symbol is not necessarily a character of text: a symbol can be any amount of data we choose, but it is often one byte's worth.

14.4.1 Entropy and Minimum Redundancy

To begin, let's revisit the concept of entropy introduced at the beginning of the chapter. Recall that every set of data has some informational content, which is called its entropy. The entropy of a set of data is the sum of the entropies of each of its symbols. The entropy *S* of a symbol *z* is defined as:

$S_z = -\lg P_z$

where *Pz* is the probability of *z* being found in the data. If it is known exactly how many times *z* occurs, *Pz* is referred to as the *frequency* of *z*. As an example, if *z* occurs 8 times in 32 symbols, or one-fourth of the time, the entropy of *z* is:

$-\lg(1/4) = 2$ bits

This means that using any more than two bits to represent *z* is more than we need. If we consider that normally we represent a symbol using eight bits (one byte), we see that compression here has the potential to improve the representation a great deal.

Table 14.1 presents an example of calculating the entropy of some data containing 72 instances of five different symbols. To do this, we sum the entropies contributed by each symbol. Using "U" as an example, the total entropy for a symbol is computed as follows. Since "U" occurs 12 times out of the 72 total, each instance of "U" has an entropy that is calculated as:

$-\lg(12/72) = 2.584963$ bits

Consequently, because "U" occurs 12 times in the data, its contribution to the entropy of the data is calculated as:

(2.584963)(12) = 31.01955 bits

In order to calculate the overall entropy of the data, we sum the total entropies contributed by each symbol. To do this for the data in Table 14.1, we have:

31.01955+36.000000+23.53799+33.94552+36.95994 = 161.46300 bits

If using 8 bits to represent each symbol yields a data size of (72)(8) = 576 bits, we should be able to compress this data, in theory, by up to:

1-(161.463000/576) = 72.0%

Table 14.1. The Entropy of a Set of Data Containing 72 Instances of 5 Different Symbols

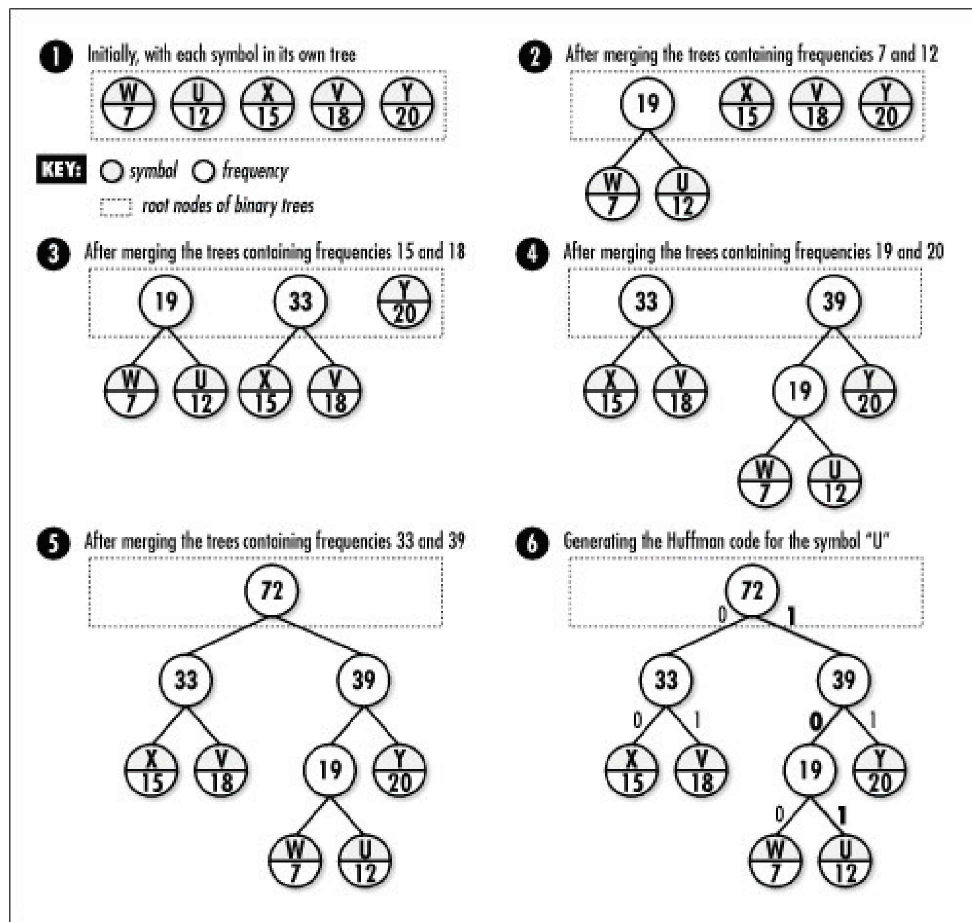
Symbol	Probability	Entropy of Each Instance	Total Entropy
U	12/72	2.584963	31.01955
V	18/72	2.000000	36.00000
W	7/72	3.362570	23.53799
X	15/72	2.263034	33.94552
Y	20/72	1.847997	36.95994

14.4.2 Building a Huffman Tree

Huffman coding presents a way to approximate the optimal representation of data based on its entropy. It works by building a data structure called a *Huffman tree*, which is a binary tree (see Chapter 9) organized to generate *Huffman codes*. Huffman codes are the codes assigned to symbols in the data to achieve compression. However, Huffman codes result in compression that only approximates the data's entropy because, as you may have noticed in Table 14.1, the entropies of symbols often come out to be fractions of bits. Since the actual number of bits used in Huffman codes cannot be fractions in practice, some codes end up with slightly too many bits to be optimal.

Figure 14.1 illustrates the process of building a Huffman tree from the data in Table 14.1. Building a Huffman tree proceeds from its leaf nodes upward. To begin, we place each symbol and its frequency in its own tree (see Figure 14.1, step 1). Next, we merge the two trees whose root nodes have the smallest frequencies and store the sum of the frequencies in the new tree's root (see Figure 14.1, step 2). This process is then repeated until we end up with a single tree (see Figure 14.1, step 5), which is the final Huffman tree. The root node of this tree contains the total number of symbols in the data, and its leaf nodes contain the original symbols and their frequencies. Because Huffman coding continually seeks out the two trees that appear to be the best to merge at any given time, it is a good example of a greedy algorithm (see Chapter 1).

Figure 14.1. Building a Huffman tree from the symbols and frequencies in Table 14.1



14.4.3 Compressing and Uncompressing Data

Building a Huffman tree is part of both compressing and uncompressing data. To compress data using a Huffman tree, given a specific symbol, we start at the root of the tree and trace a path to the symbol's leaf. As we descend along the path, whenever we move to the left, we append to the current code; whenever we move to the right, we append 1. Thus, in Figure 14.1, step 6, to determine the Huffman code for "U" we move to the right (1), then to the left (10), and then to the right again (101). The Huffman codes for all of the symbols in the figure are:

U = 101, V = 01, W = 100, X = 00, Y = 11

To uncompress data using a Huffman tree, we read the compressed data bit by bit. Starting at the tree's root, whenever we encounter in the data, we move to the left in the tree; whenever we encounter 1, we move to the right. Once we reach a leaf node, we generate the symbol it contains, move back to the root of the tree, and repeat the process until we exhaust the compressed data. Uncompressing data in this manner is possible because Huffman codes are *prefix free*, which means that no code is a prefix of any other. This ensures that once we encounter a sequence of bits that matches a code, there is no ambiguity as to the symbol it represents. For example, notice that 01, the code for "V," is not a prefix of any of the other codes. Thus, as soon as we encounter 01 in the compressed data, we know that the code must represent "V."

14.4.4 Effectiveness of Huffman Coding

To determine the reduced size of data compressed using Huffman coding, we calculate the product of each symbol's frequency times the number of bits in its Huffman code, then add them together. Thus, to calculate the compressed size of the data presented in Table 14.1 and Figure 14.1, we have:

(12)(3)+(18)(2)+(7)(3)+(15)(2)+(20)(2) = 163 bits

Assuming that without compression each of the 72 symbols would be represented with 8 bits, for a total data size of 576 bits, we end up with the following compression ratio:

1-(163/576)=71.7%

Once again, considering the fact that we cannot take into account fractional bits in Huffman coding, in many cases this value will not be quite as good as the data's entropy suggests, although in this case it is very close.

In general, Huffman coding is not the most effective form of compression, but it runs fast both when compressing and uncompressing data. Generally, the most time-consuming aspect of compressing data with Huffman coding is the need to scan the data twice: once to gather frequencies, and a second time actually to compress the data. Uncompressing the data is particularly efficient because decoding the sequence of bits for each symbol requires only a brief scan of the Huffman tree, which is bounded.

14.5 Interface for Huffman Coding

huffman_compress

```
int huffman_compress(const unsigned char *original, unsigned char **compr
   int size);
```

Return Value

Number of bytes in the compressed data if compressing the data is successful, or -1 otherwise.

Description

Uses Huffman coding to compress a buffer of data specified by `original` , which contains `size` bytes. The compressed data is written to a buffer returned in `compressed` . Since the amount of storage required in `compressed` is unknown to the caller, *huffman_compress* dynamically allocates the necessary storage using *malloc* . It is the responsibility of the caller to free this storage using *free* when it is no longer needed.

Complexity

O (*n*), where *n* is the number of symbols in the original data.

huffman_uncompress

```
int huffman_uncompress(const unsigned char *compressed, unsigned
   char **original);
```

Return Value

Number of bytes in the restored data if uncompressing the data is successful, or -1 otherwise.

Description

Uses Huffman coding to uncompress a buffer of data specified by *compressed* . It is assumed that the buffer contains data previously compressed with *huffman_compress* . The restored data is written to a buffer returned in *original* . Since the amount of storage required in *original* may not be known to the caller, *huffman_uncompress* dynamically allocates the necessary storage using *malloc* . It is the responsibility of the caller to free this storage using *free* when it is no longer needed.

Complexity

O (*n*), where *n* is the number of symbols in the original data.

14.6 Implementation and Analysis of Huffman Coding

With Huffman coding, we try to compress data by encoding symbols as Huffman codes generated in a Huffman tree. To uncompress the data, we rebuild the Huffman tree used in the compression process and convert each code back to the symbol it represents. In the implementation presented here, a symbol in the original data is one byte.

14.6.1 huffman_compress

The *huffman_compress* operation (see Example 14.3) compresses data using Huffman coding. It begins by scanning the data to determine the frequency of each symbol. The frequencies are placed in an array, `freqs` . After scanning the data, the frequencies are scaled so that each can be represented in a single byte. This is done by determining the maximum number of times any symbol occurs in the data and adjusting the other frequencies accordingly. Since symbols that do not occur in the data should be the only ones with frequencies of 0, we perform a simple test to ensure that any nonzero frequencies that scale to less than 1 end up being set to 1 instead of 0.

Once we have determined and scaled the frequencies, we call *build_tree* to build the Huffman tree. The *build_tree* function begins by inserting into a priority queue one binary tree for each symbol occurring at least once in the data. Nodes in the trees are `HuffNode` structures (see Example 14.1). This structure consists of two members: `symbol` is a symbol from the data (used only in leaf nodes), and `freq` is a frequency. Each tree initially contains only a single node, which stores one symbol and its scaled frequency as recorded and scaled in the `freqs` array.

To build the Huffman tree, we use a loop to perform `size` - 1 merges of the trees within the priority queue. On each iteration, we call *pqueue_extract* twice to extract the two binary trees whose root nodes have the smallest frequencies. We then sum the frequencies, merge the trees into a new one, store the sum of the frequencies in the new tree's root, and insert the new tree back into the priority queue. We continue this process until, after `size` - 1 iterations, the only tree remaining in the priority queue is the final Huffman tree.

Using the Huffman tree built in the previous step, we call *build_table* to build a table of the Huffman codes assigned to every symbol. Each entry in the table is a `HuffCode` structure. This structure consists of three members: `used` is a flag set to 1 or indicating whether the entry has a code stored in it, `code` is the Huffman code stored in the entry, and `size` is the number of bits the code contains. Each code is a short integer because it can be proven (although this is not shown here) that when all frequencies are scaled to fit within one byte, no code will be longer than 16 bits.

We build the table by traversing the Huffman tree using a preorder traversal (see Chapter 9). In each activation of *build_table* , `code` keeps track of the current Huffman code being generated, and `size` maintains the number of bits it contains. As we traverse the tree, each time we move to the left, we append to the code; each time we move to the right, we append 1. Once we encounter a leaf node, we store the Huffman code into the table of codes at the appropriate entry. As we store each code, we call the network function *htons* as a convenient way to ensure that the code is stored in big-endian format. This is the format required when we actually generate the compressed data in the next step as well as when we uncompress it.

While generating the compressed data, we use `ipos` to keep track of the current byte being processed in the original data, and `opos` to keep track of the current bit we are writing to the buffer of compressed data. To begin, we write a header that will help to rebuild the Huffman tree in *huffman_uncompress* . The header contains a four-byte value for the number of symbols about to be encoded followed by the scaled frequencies of all 256 possible symbols, including those that are 0. Finally, to encode the data, we read one symbol at a time, look up its Huffman code in the table, and write each code to the compressed buffer. We allocate space for each byte in the compressed buffer as we need it.

The runtime complexity of *huffman_compress* is $O(n)$, where n is the number of symbols in the original data. Only two parts of the algorithm depend on the size of the data: the part in which we determine the frequency of each symbol, and the part in which we read the data so we can compress it. Each of these runs in $O(n)$ time. The time to build the Huffman tree does not affect the complexity of *huffman_compress* because the running time of this process depends only on the number of different symbols in the data, which in this implementation is a constant, 256.

14.6.2 huffman_uncompress

The *huffman_uncompress* operation (see Example 14.3) uncompresses data compressed with *huffman_compress* . This operation begins by reading the header prepended to the compressed data. Recall that the first four bytes of the header contain the number of encoded symbols. This value is stored in `size` . The next 256 bytes contain the scaled frequencies for all symbols.

Using the information stored in the header, we call *build_tree* to rebuild the Huffman tree used in compressing the data. Once we have rebuilt the tree, the next step is to generate the buffer of restored data. To do this, we read the compressed data bit by bit. Starting at the root of the Huffman tree, whenever we encounter a bit that is in the data, we move to the left; whenever we encounter a bit that is 1, we move to the right. Once we encounter a leaf node, we have obtained the Huffman code for a symbol. The decoded symbol resides in the leaf. Thus, we write this symbol to the buffer of restored data. After writing the symbol, we reposition ourselves at the root of the tree and repeat the process. We use `ipos` to keep track of the current bit being processed in the compressed data, and `opos` to keep track of the current byte we are writing to the buffer of restored data. Once `opos` reaches `size` , we have regenerated all of the symbols from the original data.

The runtime complexity of *huffman_uncompress* is $O(n)$, where n is the number of symbols in the original