Huffman Encoding

Huffman Encoding is a method to reduce the number of bits required to store your data. To achieve this, it builds a frequency table and uses that to build a Huffman tree to decide the codes to use for each value.

The only real requirement is that each of our codes are uniquely identifiable from each other. Building the Huffman tree allows us to guarantee that each of our codes are unique enough. We need a frequency table to build the tree.

Let’s use the string ABAABACBCCBABC

First, build a frequency table

|  |  |
| --- | --- |
| Value | Frequency |
| A | 5 |
| B | 5 |
| C | 4 |

Next, build the Huffman tree. We will do this in steps.

First, order by frequency so that the greater frequencies are on the left

C:4

B:5

A:5

Next, group the two least from the right

C:4

B:5

A:5

frequency:9

Repeat last step until you reach the original max.

frequency:14

C:4

B:5

A:5

frequency:9

Lastly, follow the paths adding a 0 when you go left and a 1 when you go right

1

0

1

0

C:4

B:5

A:5

Now we make a table with the final Huffman codes

|  |  |
| --- | --- |
| Value | Huffman Code |
| A | 0 |
| B | 10 |
| C | 11 |

Let’s store our string with these codes to see the size.

For reference: ABAABACBCCBABC

0 10 0 0 10 0 11 10 11 11 10 0 10 11

01000100 11101111 10010110

(\*Note: red text indicates padding)

The final output:

0x44 0xEF 0x96

The original output:

0x65 0x66 0x65 0x65 0x66 0x65 0x67 0x66 0x67 0x67 0x66 0x65 0x66 0x67

We have saved a considerable amount of space due to the lack of different data we use as well as the frequency at which they appear. Instead of 14 bytes, we use 3 bytes. We do need to store the table however which will require more space. In this case, it requires at least 6 bytes. In a real-world case, you could provide the frequency, so they could build the Huffman tree or some way of establishing how many bits to read.

Also note that just like LZW compression, the codes refer to an index in a dictionary. This means that you can decode with out the dictionary (Huffman tree in this case) into an intermediate form that can be dealt with later if each of the values are uniquely identifiable provided you know the max size of a code

Canonical Huffman Code changes our codes to values that could be identified without a dictionary. It also benefits from storage since you only must specify the bit-length and the value for it.

In the standard method, you must store the value, bit-length, and the code. In Canonical Huffman Code, the code is implied.

Canonical Huffman Encoding

Canonical Huffman Encoding can benefit from the saved space that the standard method does but allows you to store the tree easily.

Trees are stored using the value and the length of the code for that value.

Example:

a, 2

b, 3

c, 1

d, 3

This example is from geeks for geeks by the way.

<https://www.geeksforgeeks.org/canonical-huffman-coding/>

The actual codes for that data set is as follows

a, 11

b, 100

c, 0

d, 101

The code that you get from Canonical Huffman Encoding is as follows

a, 10

b, 110

c, 0

d, 111

The approach is simple. First, sort the values by bit length like so

c, 1

a, 2

b, 3

d, 3

Next, assign the first value with 0 with the necessary 0s in the code

c gets 0

Next, add one to the previous code. If the next value has a bit length greater than the previous, add 0s to the end of the code till it is the correct length.

a gets 10

We added 1 to 0 and then put a 0 behind it

Repeat the process

b gets 110

d gets 111

b and d have the same bit length, so no 0s were added to the end. The plus 1 still occurs though.

That is Canonical Huffman coding simply put. If the values are already known, then you can just store lengths and assign them accordingly.

Note that the order of sorting is important. When sorting the data, use a stable algorithm.

One that keeps the relative order of elements.