Group HW #5 - Deep Dive into Huffman Coding

Duncan Mazza, Gati Aher, Robin Graham-Hayes, Alex Butler

October 25, 2021

1 Introduction to Huffman Coding

Huffman coding is a variation on prefix codes to help optimize lossless data compression. We present an overview of Huffman coding, followed by a variety of questions whose answers provide insight into Huffman coding and related information theory, graph theory, and more.

1.1 Prefix codes

Prefix-free coding is a scheme that makes use of specially constructed binary trees (known as prefix trees int his context) to encode data using by concatenating codewords that are constructed such that no codeword is a prefix of another codeword.

Consider the example shown in figure 1 where a prefix tree is shown for alphabet of symbols A, E, M, N, S, and T. Each edge can either be a 1 or 0, and by convention, every left edge is assigned a 0. By following a path from the tree's root to a leaf, the edge labels along the way give the prefix-free code for that leaf. For instance the letter A is encoded as 110.

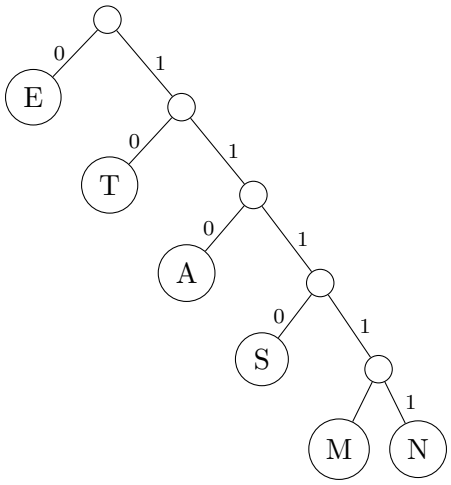

Figure 1: Example of alphabetic prefix codes

Extending this idea, we can encode any word containing these symbols using this method. Because of the prefix-free nature of the codes, the codewords can be concatenated and the original message reconstructed when parsing the bitstream from left to right. Consider as an example the encoding of the word "Statement": 1110, 10, 110, 10, 0, 11110, 0, 11111, 10 = 11101011010011110011111110. Using this prefix code we were able to encode the word statement in 25 bits.

1.2 Huffman Coding

Huffman codes takes the idea of prefix-free coding and makes it optimal. By taking the frequency of the parts of what we want to encode, an optimal prefix tree can be constructed. If we used prefix codes without taking into account frequency of the encoded characters, we could end up using more bits to encode something. For instance in figure 1 if we switched N and T on the tree, the word statement (T has the highest frequency) would take up 31 bits.

For illustrative purposes, let us again encode the word "Statement" using a different prefix tree constructed using Huffman's algorithm. Note the frequencies of each letter in the word: T(3), E(2), A(1), M(1), N(1), and S(1). The algorithm begins by creating a forest of trees where each symbol is the child of a root node containing its frequency. Then, the algorithm iteratively combines the trees in least-to-greatest order according to the sum of each tree's leaves' frequencies. Figure 2 shows the forest after a couple iterations of the algorithm. Note how the interior nodes of the trees contain the sum of their leaves' frequencies. We can see a final Huffman tree for the word "Statement" in figure 3.

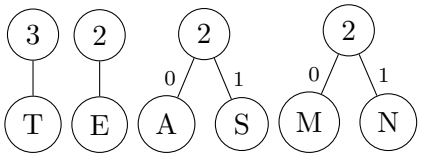

Figure 2: In-progress assembling of a prefix tree using Huffman's algorithm.

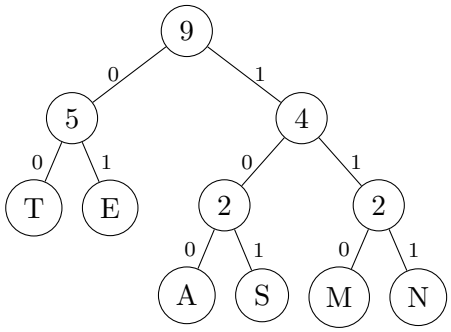

Figure 3: Completed prefix tree created from Huffman's algorithm.

Using this tree, the word "Statement" can be encoded as: 101, 00, 100, 00, 01, 110, 01, 111, 00 = 1010010000011100111100, with n=22 bits.

By using Huffman coding we were able to reduce the word "Statement" to 22 bits. In comparison, if each of the 6 letters were assigned a binary value (000 to 101), the word statement would be 27 bits long. If we were encoding an entire book or database, these differences would add up to a considerable amount of additional storage space.

1.3 Variations of Canonical Huffman Coding

While Huffman coding on its own is extremely useful, there are many options to modify how Huffman coding works in order to deal with specific circumstances or to optimize the coding further.

1.3.1 Adaptive Huffman

Adaptive Huffman coding was developed to optimize Huffman codes in every scenario. These codes encode the message as it is being sent, altering the tree as each character is processed. At the beginning of encoding the tree begins with only one node (traditionally labeled NYT [Not Yet Transferred]). Each space on the tree is pre-numbered, and all numbers in the tree must be in descending order. As a letter is processed, it is added to the tree. If it already exists on the tree, its frequency is updated, and will move along the tree if necessary.

1.3.2 DEFLATE (PKZIP)

DEFLATE (originally referred to as PKZIP) is an existing, and slightly deprecated, application of adaptive Huffman coding used in compression today. Released in 1996, DEFLATE was created by Phil Katz as a lossless data compression algorithm. Using a combination of an LZ77 algorithm and adaptive Huffman coding, the algorithm is as a whole is greater than the sum of its parts: by working together each component of DEFLATE is able to succeed. The adaptive Huffman coding creates an efficient tree of codes, based on the sequence of data in the alphabet provided, replacing commonly used characters/symbols, and creating compressed blocks. Then the LZ77 compression algorithm is applied over the Huffman coding output, replacing identical strings of bits with a reference to the first incident of the string in the code. Additionally, there is other functionality baked into DEFLATE to allow for easy use, such as compressing with fixed Huffman codes. [10] [3]

1.3.3 N-ary Huffman

Huffman's original encoding was N-ary, so instead of being purely binary, the tree could have any number or branches at each junction. With N-ary Huffman, the n characters with the lowest weights are taken instead of just two. However, with N-ary Huffman sometimes filler characters with 0 frequency must be added to the tree to ensure the tree can function.

1.4 Why is Huffman used?

Huffman coding is used so extensively because of two main features. Firstly, it is straightforward to develop an efficient implementation. Secondly, many other lossless compression techniques, such as *Arithmetic coding*, fall under patents. Because Huffman was unable to patent his student work, it has made his coding readily available to start and build off of.

2 How far can Huffman compression go?

In §1.3.2, a compression scheme was introduced where the output of Huffman coding was run through another compression. The curious reader may wonder: how many times can one compress the output of the compression? What are the limits of Huffman coding? An answer to this question requires the introduction of two foundational concepts from information theory: self-information (also known as surprisal) and entropy. These concepts are then applied in the context of Huffman coding and used to illustrate the limits of lossless compression.

2.1 Self-information

Information theory, as developed by several prominent theorists and engineers (including Claude E. Shannon) over the early 1900s [1], states that the amount of information I contained by an event x_i of probability p_i (i.e., its self-information) is given by:

$$I(x_i) = -\log(p_i) = \log\left(\frac{1}{p_i}\right) \text{ bits.} \tag{1}$$

The units of information is determined by the base of the logarithm. Unless otherwise stated, assume a base-2 logarithm is used such that the unit of information is bits.

Note that $\log(0)$ and $\log(1)$ evaluate to ∞ and 0 respectively, corresponding to improbable and guaranteed events containing infinite information, respectively. For further illustration, consider the information contained by the result of a coin flip:

$$I(\text{heads}) = -\log(0.5) = -\log\left(2^{-1}\right) = \log(2) = 1 \text{ bit}. \tag{2}$$

2.2 Entropy

In information theory, the entropy of a discrete random variable X is defined as the expected value of the self-information the random variable. To understand what is meant by "expected value" (which has broad applications far beyond information theory), it can be informally defined as the weighted average. Formally, the expected value of a function f of a random discrete variable X is [8]:

$$E[f(X)] = \sum_{i=1}^{N} p_i f(x_i) \tag{3}$$

where $E[X]$ denotes the expected value of X. Therefore, the entropy of X is given by:

$$H(X) = E[I(X)] = -\sum_{i=1}^{N} p_i \log(p_i) \tag{4}$$

where $H(X)$ denotes the entropy of X.

2.3 Self-information and Entropy as Applied to Huffman Coding

Consider the implication of entropy as it applies to coding: to encode a symbol $x_i \in X$, at least $I(x_i)$ bits of information are needed. The minimum average code length for a symbol X is then $H(X)$. This leads to a formula for the efficiency η of the Huffman coding algorithm:

$$\eta = \frac{H(X)}{\overline{L}} \tag{5}$$

where $\overline{L}$ is the average codeword length[1]. Through this lens, then, Huffman coding can be viewed as an algorithm that intelligently assigns codes to symbols such that $\overline{L}$ is minimized[1], with a globally optimal solution's average codeword length being bound by $H(X)$[5].

This theoretical lens neglects an important consideration when computing the amount of bits needed to store information. For example, one could arbitrarily the symbol ℵ to represent the bitstream 00110100. The alphabet X has only one symbol, meaning that the probability of ℵ is 1 and its self-information is zero. The entropy of X then becomes 0, and the bitstream can be compressed using 0 bits of data. Clearly, the means by which symbols are mapped to their codewords must also be considered.

Compression is all about removing redundancy from a set of information. Though Huffman coding removes most if not all redundancy from the input data, the inclusion of the lookup table/binary tree used to decode the compressed data may introduce some amount of redundancy that a subsequent compression could take advantage of. Regardless, it is important to note that for any given compression scheme, the no two inputs can compress to the same output. Within this constraint, Huffman coding is extremely effective and provides near-optimal compression from only one pass.

[1]It can be proven using proof-by-induction that Huffman's algorithm produces an optimal prefix tree. Note that this optimization is with respect to a particular input, where the frequencies of symbols in the source determine what is considered optimal.

3 Error Correction and Resiliance: What resilience do Huffman Codes have to errors?

Errors can occur in all layers of data storage. To counter this, most stages that encounter high rates of induced error (such as transport) rely on redundancy to encode the data. However when it comes to compressed data, which theoretically has the least amount of redundancies, a single error can multiply when uncompressed into many. This understanding of the effects of errors on data extends to data that is encoded using Huffman coding. The question comes to mind: *How do we correct for error in compressed data, and what systems exist already in Huffman Coding to minimize the effects of error?* There exist emergent properties of error resilience in Huffman coding that make it especially interesting, such as re-synchronization. To understand this concept we will introduce types of errors that Huffman codes may encounter, an example of the effects of an error on compressed data, and move into the property of synchronization.

3.1 Types of Errors

When understanding errors in data there are five types of errors that are typically defined:

- Bit-Flip Errors: one bit's value changes from a 1 to a 0 or a 0 to a 1
- Bit Insertion Errors: an additional bit is added into the string (0 or 1)
- Bit deletion Errors: a bit is removed from the string
- Burst Errors: consecutive bits are altered and corrupted
- Missing Fragments: a chunk of data is missing

Any one or many of these errors can possibly occur while our data is compressed, and all of them will radically alter data when uncompressed.

3.2 Examples of Errors in Huffman Codes

To show the effect a single bit error would have across a simple implementation of a Huffman code, we will create a Huffman code called H defined as:

$$a \to 00$$
$$b \to 01$$
$$c \to 10$$
$$d \to 110$$
$$e \to 111$$

Next we will encode a given string S, which we will define as $S = bbeaaebcec$ with H, giving us the string:

$$H(S) = 01011110000111011011110.$$

Now, to simulate a one bit error, the fifth bit in the string is flipped from a 1 to 0, giving the string S_E:

$$H(S_E) = 01010110000111011011110.$$

We then use our Huffman code from earlier (H) to decode S_E, giving:

$$H^{-1}H(S_E) = bbcabddec.$$

At face value the output code with a single error has little value. Only one bit flipped yet it resulted in a dramatic difference between S & S_E. Specifically, the one bit change only flipped a bit in the first e of the original code, but changed five characters in S by creating an offset for the rest of the code down the line. Notably, every bit in S was not completely incorrect after this change, as the last two characters were unchanged from this single error. In the case of this error, we can see that the code went from a state of synchronization (sync), where the decoder is in sync with the codewords resulting in every bit decoded successfully, to a region of unsynchronization (unsync), where the decoder begins to use components of other codewords to decode the incorrect word. This region was sparked by the single bit-flip error. After this period of unsync, the code can be seen back in synchronization. In fact, if we were to extend the string S, the error would continue to be completely contained within those five characters that were corrupted.

3.2.1 Resynchronization & Resilience of Huffman Codes using Synchronizing Codewords

This phenomenon within Huffman Codes is called resynchronization, and its an interesting property of Huffman codes. While the error is responsible for the unsync, a grouping of codewords in S is responsible for the resync. Perkins and Escott wrote extensively about this in their 1998 paper "Synchronizing codewords of q-ary Huffman codes" [7]. To identify a key concept they introduced, note in our example above how the bit-string 0110 (bits 15-18) acts as a synchronizing string. What is meant by a "synchronizing string" is that after the decoder is feed the original string S and string with error S_E, any error propagation before a synchronizing codeword (SC) will resolve after the decoder finishes the SC.

Some algorithms deliberately use SC's as a way to minimize error[2]; however, SC's can be improved upon greatly. A great example of this can be seen in our example from earlier, where removing instead of flipping the bit results in $H^{-1}H(S_E) = bbdabddec$, which is one digit shorter than the original string S. While preventing the error from propagating is useful, losing the positions of our data is a major loss, which extended synchronizing codewords (ESCs) aim to solve [6]. Developing ESCs takes a lot of overhead when designing a code, since you must effectively code a specific property into your code, whereas SC's are already existing properties of Huffman codes as we known them.

4 MP3 Stenography: How are Huffman tables used to communicate secret messages?

4.1 Huffman Tables in MP3

Turning raw audio into an MP3 file involves a quantization-based lossy compression and a Huffman-based lossless compression. For each audio segment, the MP3 compression algorithm finds the optimal quantization step size, scaling coefficients, and Huffman table choice to minimize both

[2] And in some cases discerning error regions from correct data, can still provide value

bitstream size and audio distortion. On average, using the Huffman codes allows the MP3 file format to save about 20% of space [4]. Since Huffman codes can shorten repetitive byte patterns, the Huffman compression works best on pure, digitized sounds.

The final MP3 output file consists of a series of frames, where each frame consists of 1152 samples. Each frame is divided into two granules, and each granule is divided into three kinds of regions: big_value region, count1 region, and rzero region. The rzero sub-regions are all zeros. The count1 region is binary. The big_value region consists of a series of numerical coefficients. Each big_value region is further divided into 3 sub-regions.

The MP3 specification defines 34 Huffman tables. Huffman table 4 and table 14 are not used. The rzero regions are not encoded. Huffman table 32 and table 33 are used to encode count1 values. The remaining 30 tables are used to encode the big_value sub-region, one per sub-region.

In the standard format, the choice of Huffman table depends on (1) the maximum value in the sub-region to encode and (2) the table that results in the minimal number of bits. The optimal Huffman table is chosen by first rejecting tables that either cannot encode the maximum value (have too low maximum value) or tables that do not efficiently encode the maximum value (have too high maximum value). Figure 4 shows how multiple tables can have the same max value. Of the remaining choices, the table that results in the smallest number of encoding bits is the optimal table choice.

Table	Max Value	Table	Max Value
0	0	16	16 (1 linbit)
1	1	17	19 (2 linbit)
2	2	18	23 (3 linbit)
3	2	19	31 (4 linbit)
4	X	20	79 (6 linbit)
5	3	21	271 (8 linbit)
6	3	22	1039 (10 linbit)
7	5	23	8207 (13 linbit)
8	5	24	31 (4 linbit)
9	5	25	41 (5 linbit)
10	7	26	79 (6 linbit)
11	7	27	143 (7 linbit)
12	7	28	271 (8 linbit)
13	15	29	527 (9 linbit)
14	X	30	2016 (11 linbit)
15	15	31	8207 (13 linbit)

Figure 4: Properties of big_value Huffman tables used in MP3 specification.

The maximum value of the binary encoding is $15 + 2^{\text{linbits}}$. Huffman tables 0 through 15 use different codebooks with no linbits, tables 16 through 23 operate in a single codebook with different numbers of linbits, and tables 24 through 31 use another codebook with different numbers of linbits.

4.2 Huffman Table Stenography

The goal of stenography is to hide and recover secret messages. Since MP3 is a ubiquitous audio format, it is an inauspicious choice for secret message transfers. A stenography scheme creates a secret compartment-esque way to hide a bitstring in a MP3 file. Encoding a secret message happens in the Huffman-based lossless compression stage, as encoding a message before that, in the lossy compression stage, might lose the secret message.

The choice of Huffman table for each big_value sub-region can be used to hide a secret bitstring. The Huffman table-swapping algorithm divides the 30 possible tables into three groups: G_0 (tables representing 0), G_1 (tables representing 1), and G_{-1} (tables that cannot be used for encoding). If

the secret bit is 0 and the optimal table is in G_0 then the table choice is left unchanged. Likewise, if the secret bit is 1 and the optimal table is in G_1 the table choice is left unchanged. However, if the optimal table group and the secret bit do not match, the table should be swapped with a corresponding table in the other group. The the new table should be chosen so that it has a maximum value greater than or equal to the original sequence's maximum value.

In 2009, Yan and Wang published a table swapping scheme [2] that used 12 tables in G_1, 13 tables in G_0 and 5 tables in G_{-1}, as seen in (a) of Figure 5. This encoding had low audio degradation and perceptibility, but as a sub-region had an increasing number of Huffman tables in the G_{-1} set the hiding capacity decreased. With this scheme, table 3 and 13 could not be used for encoding because they would swap with 4 and 14 respectively, which are unused. Table 23 cannot be used because table 23 (max value of 8207) cannot be swapped with table 24 (max value of 31). Table 31 could not be used because table 32 is used to encode a different region.

In 2019, Tan et. al. published an improved table swapping scheme [9] that used 15 tables in G_1, 14 tables in G_0 and just 1 table in G_{-1} (see (b) of Figure 5). By reducing the number of unused tables, the new encoding achieved a much higher capacity. While this scheme experienced minimal audio degradation, the encoding was more detectable, not only because it encoded more data, but also because it changed the probability distributions of table 15. In the standard process, table 15 has a higher probability of being chosen because of its peculiar positioning between the non-linbit and linbit-using tables, but the proposed swapping method results in equal probability distributions for table 13 and table 15.

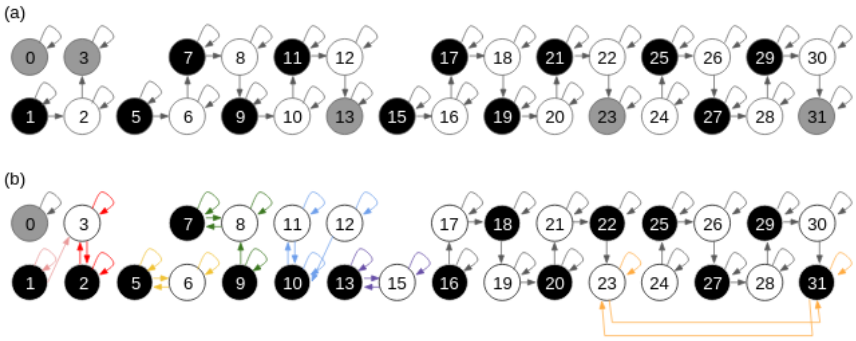

Figure 5: Nodes are tables, directed edges represent swaps. G_1 tables are colored black, G_0 tables are colored white, and G_{-1} tables are colored gray. (a) 2009 table swapping strategy (b) 2019 table swapping strategy

References

[1] Marcelo S. Alencar. *Information Theory*. Communications and Signal Processing Collection. Momentum Press, 2015. ISBN: 9781606505281. URL: http://olin.idm.oclc.org/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=880325&site=eds-live.

[2] R. Wang D. Yan. "Huffman table swapping-based steganograpy for MP3 audio". In: *Multimedia Tools and Applications* 52 (2011), pp. 291–305. DOI: 10.1007/s11042-009-0430-5.

[3] L. Peter Deutsch. *DEFLATE Compressed Data Format Specification version 1.3*. RFC 1951. May 1996. DOI: 10.17487/RFC1951. URL: https://rfc-editor.org/rfc/rfc1951.txt.

[4] Bouvigene Gabriel. *Overview of the MP3 techniques*. MP3' Tech. URL: http://www.mp3-tech.org/tech.html (visited on 10/23/2021).

[5] Venkatesan Guruswami. *15-359: Probability and Computing*. 2009. URL: https://www.cs.cmu.edu/~venkatg/teaching/ITCS-spr2013/notes/15359-2009-lecture25.pdf (visited on 10/24/2021).

[6] Wai-Man Lam and S.R. Kulkarni. "Extended synchronizing codewords for binary prefix codes". In: *IEEE Transactions on Information Theory* 42.3 (1996), pp. 984–987. DOI: 10.1109/18.490560.

[7] S. Perkins and A. E. Escott. "Synchronizing Codewords of q-Ary Huffman Codes". In: *Discrete Math.* 197–198 (Feb. 1999), pp. 637–655. ISSN: 0012-365X. DOI: 10.1016/S0012-365X(98)00268-4. URL: https://doi.org/10.1016/S0012-365X(98)00268-4.

[8] Miller Scott and Childers Donald. *Probability and Random Processes : With Applications to Signal Processing and Communications*. Vol. 2nd ed. Academic Press, 2012. ISBN: 9780123869814. URL: http://olin.idm.oclc.org/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=453841&site=eds-live.

[9] D. Tan et al. "High capacity reversible data hiding in MP3 based on Huffman table transformation". In: *Mathematical Biosciences and Engineering* 16 (2019), pp. 3183–3194. DOI: 10.3934/mbe.2019158.

[10] J. Ziv and A. Lempel. "A universal algorithm for sequential data compression". In: *IEEE Transactions on Information Theory* 23.3 (1977), pp. 337–343. DOI: 10.1109/TIT.1977.1055714.