Collaborative Deep Dive on Graph Theory - Preliminary Deliverable

Milo Song-Weiss, Trevor Zou, Philip Post, Ale Cuevas

Huffman Coding for Image Compression

Introduction

Huffman encoding is a method of compressing a file without losing any of the information it contains (i.e. lossless compression). It does this based on the principle that we often use too many bits to represent a character. As a character is digitally represented as eight bits in sequence, creating a byte, whose combination can create any value between 0 to 255 (i.e. 256 potential characters). Rarely, will you ever have a document that has all 256 potential characters, in which case Huffman encoding can be applied. For example, take the case where we have a nonsensical message purely composed of A, B, and C. Using the standard ASCII character mapping, these characters would normally be represented as follows[1]:

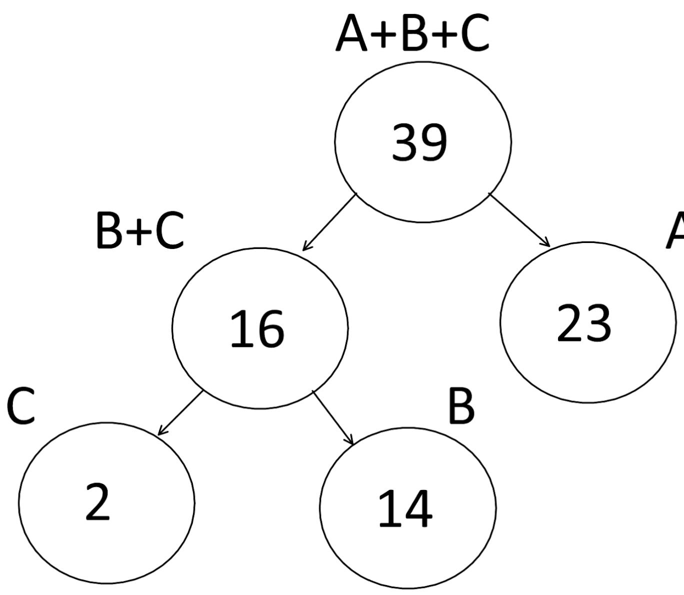
$$A = 01000001 = 0 * 2^7 + 1 * 2^6 + 0 * 2^5 + 0 * 2^4 + 0 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0 = 65$$

$$B = 01000010 = 0 * 2^7 + 1 * 2^6 + 0 * 2^5 + 0 * 2^4 + 0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0 = 66$$

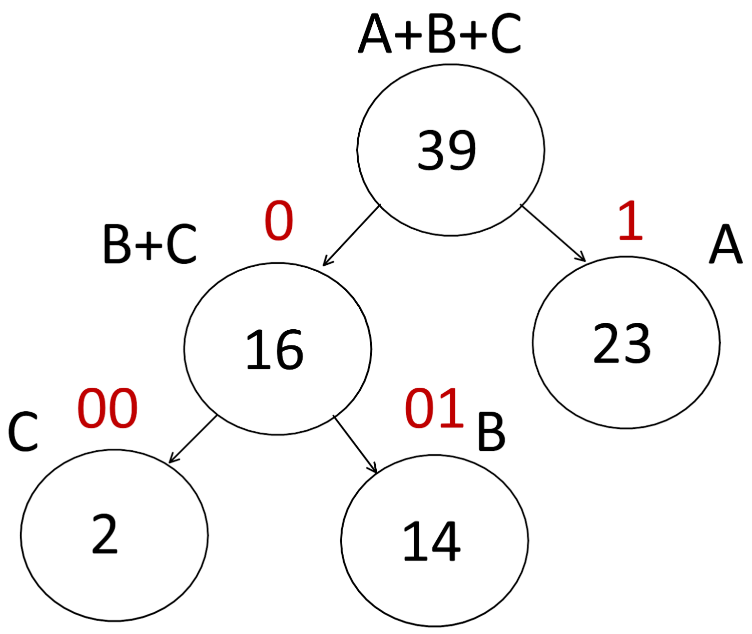
$$C = 01000011 = 0 * 2^7 + 1 * 2^6 + 0 * 2^5 + 0 * 2^4 + 0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = 67$$

But, wait. If our document only has three characters that are ever used, then why are we using eight bits for every single character? Huffman encoding addresses this excess by asking, “What is the minimum number of bits we can use for each of our characters?” For our example, let’s say A appears 23 times, B appears 14 times, and C only appears 2 times in a document we wish to compress.

[1] https://www.ascii-code.com/



We take each character and its corresponding frequency and use this information to make a Huffman tree. A Huffman tree works by creating a leaf, also referred to as a root, for the sum of the frequencies of each character. We then methodically split the larger leaves into two smaller leaves which can either be a single character or a combination of characters. The smaller of the two leaves will always branch to the left of the parent leaf. We are finished once we have a single leaf corresponding to each character as seen above. Now, it is time to assign a shorthand binary representation for each of the characters:



With each leaf, we assign progressively larger, unique binary codes for each leaf. The idea is that the most frequent characters, such as A, will get the shortest binary possible; thus, maximizing the amount of bits we are able to save. Anytime, we have a leaf that is a combination of two or more characters, the parent leaf's binary code will carry on the descendants with either a 0 or 1 appended. When we look to our example, we can see how many bits this compression would have saved us:

$$1\ bit\ length\ *\ 23\ instance\ of\ A\ =\ 23\ bit\ length$$

$$2\ bit\ length\ *\ 14\ instance\ of\ B\ =\ 28\ bit\ length$$

$$2\ bit\ length\ *\ 2\ instance\ of\ C\ =\ 4\ bit\ length$$

$$Compressed\ \ Total\ bit\ length\ =\ 55\ bit\ length$$

$$8\ bit\ length\ *\ 23\ instance\ of\ A\ =\ 184\ bit\ length$$

$$8\ bit\ length\ *\ 14\ instance\ of\ B\ =\ 112\ bit\ length$$

$$8\ bit\ length\ *\ 2\ instance\ of\ C\ =\ 16\ bit\ length$$

$$Uncompressed\ \ Total\ bit\ length\ =\ 312\ bit\ length$$

In short, we observe a space savings of around 5 times! Of course, we would also have to encode what bit combination we decided to shorthand for each character, but that would take only a few tens of more bits, still yielding a significant amount of space savings over uncompressed data.

But, how can this be applied to images? Well, images are actually very similar to character strings. Similar to how a character is symbolized by a number ranging from 0 to 255, a pixel within a grayscale image corresponds to an intensity value between 0 and 255. In reality, pixels in images are commonly encoded and read as arrays of characters for this reason, before being converted to their corresponding colors. Therefore, to apply a Huffman encoding to an image we just need to treat every pixel value as a character and then repeat the process we did above. The only other difference is that we will also have to encode a little more information about the image's dimensions so that when we decode, we know how many pixels should be in each line in the image.

Existing literature establishes three useful metrics for ascertaining how effective a compression algorithm performs[2]. The first of which is the Compression Ratio (CR), which is the ratio between the size of the original image divided by size of the compressed image:

[2] https://ieeexplore.ieee.org/document/7566549

$$Compression\ Ratio\ = \frac{Original\ Image\ Size}{Compressed\ Image\ Size}$$

The higher the compression ratio the better, as this means our compressed image was proportionally smaller than the original[3]. However the compression ratio only indicates how much the original size was reduced and gives no indication of the quality of the reduction (i.e. how accurate is the compressed image compared to the original)[4]. For this, we have Mean Square Error (MSE), which averages the difference between pixel values for every pixel in the original MxN image versus the uncompressed image:

$$Mean\ Squared\ Error\ = \frac{1}{MN} * \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} [d(i,j)\ -\ f(i,j)]^2$$

With M being the width of the image, N being the height of the image, f(i,j) being the mapping of the original image, and d(i,j) being the uncompressed version of the image. In short, this function shifts pixel by pixel, takes the difference between their values, and then averages this difference sum across all pixels. We can use Mean Squared Error to compare the efficacy of different compression methods, however, MSE is not directly applicable to Huffman Coding since it is a lossless compression method.

Lastly, we have the Bits per Pixel (BPP) of an image, which tells us the average number of bits we need to represent a single pixel of an image[5].

$$Bits\ per\ Pixel\ = \frac{Number\ of\ Bits\ in\ a\ Compressed\ Image}{Total\ Number\ of\ Pixels\ in\ an\ Image}$$

Unlike compression ratio, which only compares a compressed image to its original, BPP gives us an objective way of comparing separate algorithms. Of course, this metric does not take into account the error of compression in these comparisons.

[3] Id.
[4] Id.
[5] Id.

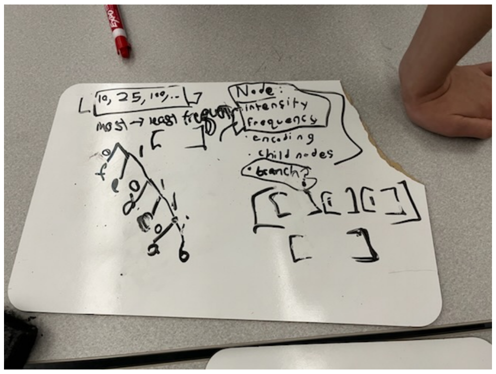
Specific plans

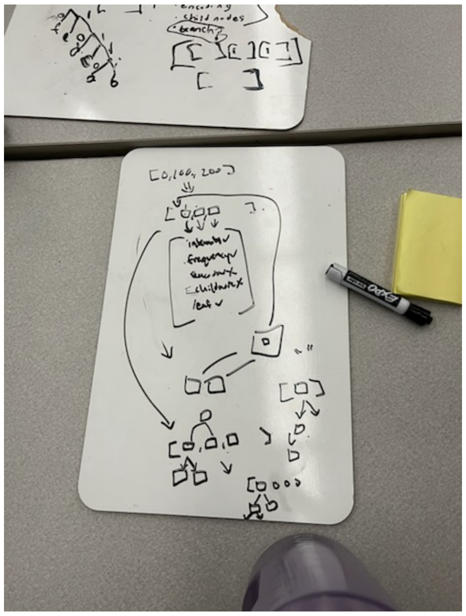
Our MVP goal is to implement, from scratch, an algorithm to compress a user-provided grayscale image and its dimensions as a binary file or binary representation. Our team is mostly interested in applications and implementations, so we decided that writing our own code to implement Huffman code suited our learning goals. We thought that Huffman coding within the context of image compression would give us an added layer of complexity without being too overwhelming. When deciding MVP vs Stretch goals, we thought that a grayscale image would provide a good base of understanding because we'd only have to work in one colorspace, but still understand how pixel intensities/frequencies could be used for the Huffman coding.

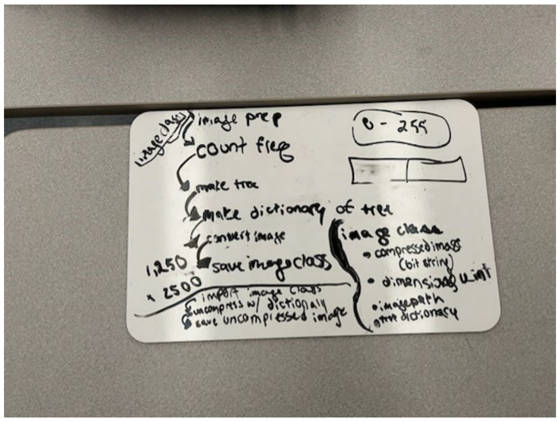
Our stretch goal is to implement, from scratch, an algorithm to compress an RGB user-provided image and its dimensions as a binary file or binary representation. Adding RGB functionality would be a viable next step after implementing Huffman Coding for grayscale images. We expect it wouldn't add too much complexity to the code, and since RGB images are the modern standard, it would add to the "useful" factor of the project. We also want to better understand how to validate our results, through both size and time comparison. Another stretch goal of ours could be to run the algorithm at scale and see whether or not the added complexity and runtime is worth it depending on how much the images vary, or whether it is faster on the whole to recreate a huffman tree for each image or use the same tree for every image on a diverse image set.

Progress and preliminary results

So far, we have each done individual learning to have a basic understanding of the theory behind huffman code, as well as compile useful resources. We then used class time to validate our own individual understanding by explaining the algorithm itself to each other. We then wrote out pseudo-code to figure out what kinds of functions and classes we'd need to have for our MVP.







We finally started our actual implementation code here:

https://colab.research.google.com/drive/1mRjzFAV4QFdSxLH_kXD_UGhAQRLzTBqu?usp=sharing

Communicating your results

For our final deliverable we will create a poster with a brief explanation of huffman coding, a walkthrough of our algorithm, and final results comparing a compressed to an uncompressed image. We will also include broader application findings, like what types of images are best suited for huffman coding. Our code will be provided in a supplemental jupyter notebook with detailed comments describing our implementation. Even though our project will in large part be code, we thought that a jupyter notebook doesn’t really communicate the theory and results as clearly as we would like, and would be overwhelming to look at on its own. We thought that a poster would provide a more concise way to understand the algorithm at a broad scale without getting bogged down in syntax.