Implementation of Huffman Coding in MATLAB with Comparison to JPEG Compression Ratios

Ryan Mia

INTRODUCTION and OBJECTIVE

As we have learned in class, Huffman Coding is one of many techniques to perform data compression. The poster presentation is two-fold.

First, I used MATLAB as a tool to create my implementation of Huffman coding. I implemented functions to generate the Huffman codes using the algorithms we learned in class, as well as encoding and decoding functions. These were used to compress and decompress images to check the compression rates we could achieve on different images as well as to verify the coding scheme worked and the decoded image was the same as the initial starting point.

The next portion of the presentation is an investigation into the JPEG Compression standard. JPEG compression is a more complicated compression technique than Huffman coding, but does use Huffman coding as a subset of its algorithm. Thus I seek to find out how JPEG compression uses Huffman coding, and how its compression rates compare to a Huffman coding compression on images, and possible tradeoffs between to two methods.

LITERATURE REVIEW

Topics covered in this presentation are Huffman Coding and JPEG Compression. This literature review will focus on the later, assuming a baseline knowledge of Huffman Coding is already understood.

JPEG stands for Joint Photographic Experts Group, a organization whose efforts focus on improving the standard for image compression. The result created the JPEG compression algorithm which is a lossy compression algorithm which relies on DCT and Quantization to compress an image in combination with Huffman Coding.

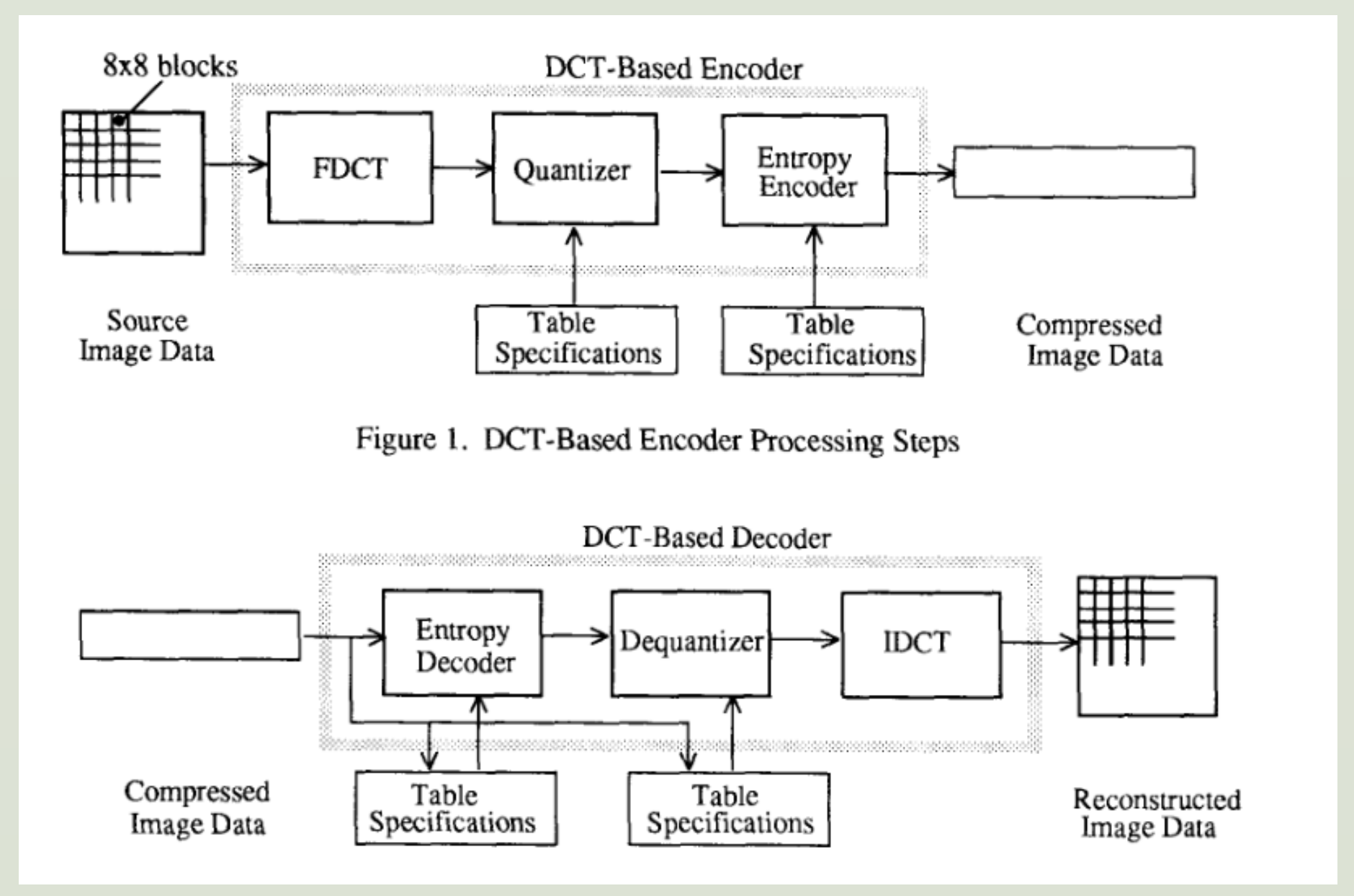


Figure 1. DCT-Based Encoder Processing Steps

The above figure shows the general steps taken in the JPEG compression. Of course this is a slightly oversimplified model, but the core steps are here. The part we are familiar with (coding) runs from step 4 (Entropy Encoder) to step 7 (Entropy Decoder). This is what I have implemented in MATLAB in the *Methods* section.

In the pre-encoding steps, the 2D DCT is applied to the image in 8x8 blocks to transform the blocks to DCT coefficients. These are similar to eigenvalues in which each coefficient represents a frequency of information. The quantization step then takes this transform and removes some small details to these coefficients by zeroing out the high frequency coefficients. These coefficients represent frequencies of detail largely unnoticeable by the human eye. This is the "lossy" step of the JPEG compression algorithm. The number of coefficients zeroed out can be surprisingly high (80% or more of the coefficients). This step is variable depending on how much loss deemed acceptable in your system. These DCT blocks are then passed to the encoder where Huffman coding is performed. Quantization allows for Huffman coding to work with data with lower entropy, thus increasing the compression rate.[1]

METHODOLOGY

Girhub Repo: https://github.com/ryanmia/Huffman_Coding_Images

get_huffocdes(): This function is the main one which uses the Huffman coding scheme learned in class. By first obtaining and sorting the pixel values in order of probability, I assure that the last 2 elements in my probability array are correspondent to the lowest two probabilities. I combine probabilities until we only have one probability (1) left. On each iteration I scan the probability array to see where the new probability could go and insert into the proper location. The complicated part of this algorithm is keeping track of which symbols are associated with each probability – especially once we start combining multiple times. This is solved by attaching an updating vector of pixels values to each probability. To avoid traversing the tree at the end of operation, I update the binary values as we go.

huff_encode(): This function takes in the dictionary we created and the image matrix reshaped to a vector. The result is a bitstream that converts each pixel value to its codeword. This involved some bitshift math to do – an oversight on my part that MATLAB does not have a 'bool' data type.

huff_decode(): This function takes in the bitstream ad the dictionary and returns the original image. Again some bitshift math is required for this operation. A benefit of the Huffman encoding algorithm is that we can check the current bit input against codewords of that length and if one matches we do not need to go any further because Huffman coding uses *prefix codes* to ensure any codeword is not a subset of another codeword.

Once it is desired to grab view the original image, we apply the steps to decompress the image. First, the decoder uses the dictionary generated by Huffman encoding and decodes the codewords back to their original form.

Next, in the post-decoder steps we start by dequantizing the DCT blocks. Dequantization is not possible to do perfectly. Sometimes this step is bypassed and left in its quantized state, and other times unique algorithms are used in attempt to dequantize as best as possible. As an example, optimal dequantization arrays can be calculated in an attempt to minimize the restoration error. This method builds upon the Miller least squares regularization solution which is a method of solving linear inverse problems. No matter what dequantization method is used, if any at all, the next step is to use and IDCT function to convert our DCT coefficient blocks back to the spatial domain. This is similar in function to performing an IFFT on frequency domain data back to time domain for those more familiar with that process.[2]

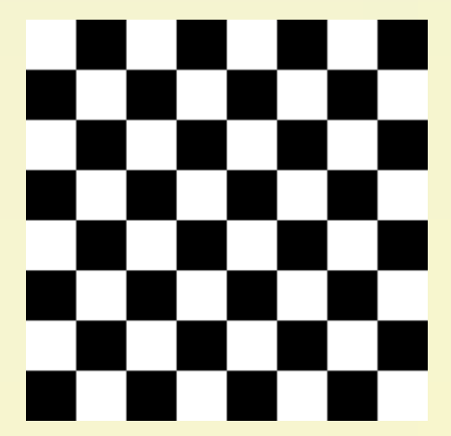
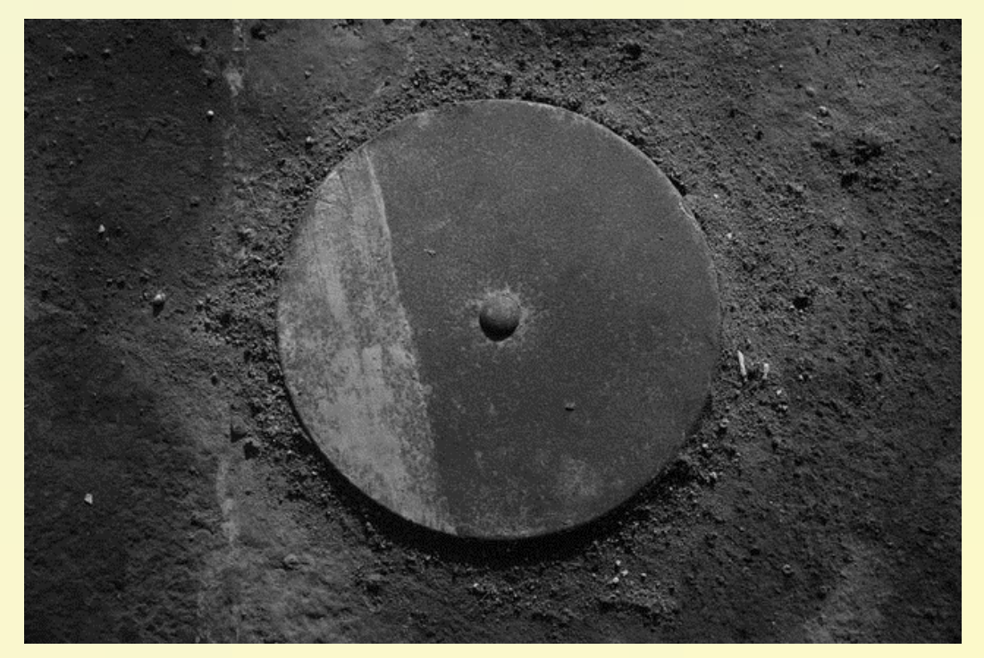
The result is the original image with minor detail loss as a product of the quantization efforts to compress the image. The amount of detail loss is dependent upon the quantization factor chosen and the performance of your dequantization algorithm. Usually this loss is barely noticeable, if at all by the human eye. One should be wary of lossy compression in systems where computer algorithms need to process the images further, as unnoticeable to the human eye does not mean a computer couldn't have used that information.

References

[1G. K. Wallace, "The JPEG still picture compression standard," in *IEEE Transactions on Consumer Electronics*, vol. 38, no. 1, pp. xviii-xxxiv, Feb. 1992, doi: 10.1109/30.125072.

[2R. Frost, Yi Ding and A. Baskurt, "JPEG dequantization array for regularized decompression," in *IEEE Transactions on Image Processing*, vol. 6, no. 6, pp. 883-888, June 1997, doi: 10.1109/83.585238.

RESULTS OF COMPRESSION ON SAMPLE IMAGES

Image	*Size*	*Huffman Compression Rate (comp. size/uncomp. size)*100*	*JPEG Compression Rate*
	2400x2400	81%	6%
	512x512	94%	12%
	200x200	24%	17%
	640x426	87%	25%
	2399x1598	93%	17%

DISCUSSION OF RESULTS

The results of the Huffman Coding and JPEG Compression on the small image set above shows a few trends that I think are insightful. First, for all images except the checker pattern (3rd) we see that Huffman Coding produces relatively small compression rates compared to the rest of JPEG Compression. This seems to align with what we have learned about JPEG compression in the *Literature Review* section. Basically, performing DCT and Quantization are more drastic measures to reduce information by a large margin. Namely, zeroing out a majority of the DCT coefficients before applying Huffman Coding allows for a huge reduction in file size as the data will be much more uniform than the majority of images. The checker pattern being an except as it already has a very uniform and low entropy of pixel values. In this case, JPEG Compression is actually doing most of the work and DCT/Quantization steps do not add a great deal of improvement.

Focusing on just the Huffman Coding rates, we can see that the images with the least variation in pixel value (i.e. checkers, and then the 4th image of the disk in the dirt) have the higher rates of compression. This aligns with what we have learned in class that these in the encoding scheme we can set the pixels with the highest probability to the shortest length codewords to reduce the average codeword length. The lower the entropy of the image, the more we can take advantage of this algorithm and reuse shorter length codewords.

A last observation that somewhat surprised me was that size of image does not play a big role in determining the compression size of an image. I included the size of the dictionary in the compressed image size as it is necessary to decode the image. But regardless of image sizes ranging from 512x512 to 2400x2400 there was not a big difference in compression rate that could be attributed to this size difference. I suspect this may not hold at very small image sizes (but then why are you even compressing at that size?), but I think this can be attributed to the fact that in a pixel range from 0-255, there are not that many codewords so the dictionary need not be that big in comparison to the bitstream of codewords. A small side note that the Huffman Encoding algorithm I created was pretty slow in MATLAB without DCT and Quantization. Due to this my computer was not able handle running many test samples of images unfortunately. Optimizations in speed by swapping to a low level programming language, using concurrent programming, and more efficiently using hash tables could be made to improve speed.