CMPT 365 Multimedia Systems

Project 2

**Q. 1.**

**Table of WAV files And Corresponding Compression Ratios (Rounded)**

|  |  |  |
| --- | --- | --- |
| File | Huffman Compression Ratio | LZW Compression Ratio |
| P2-sample\_1.wav | 1.155 | 0.945 |
| P2-sample\_2.wav | 1.188 | 0.992 |
| Q2-sample\_1.wav | 1.659 | 0.854 |
| Q2-sample\_2.wav | 1.033 | 0.832 |

Compared to FLAC which according to online research is lossless and can even go down to about half the size of the original audio source, my compression ratios for both Huffman and LZW are not as good as it (FLAC). The reason why this is the case is because FLAC incorporates multiple compression techniques such as “linear prediction, Golomb-Rice coding, and lastly, run length coding” according to the source <https://wiki.audacityteam.org/wiki/FLAC>.

Of course, out of those three fancy techniques mentioned only “run length coding” was discussed in lecture so from what I can understand this type of technique is actually quite powerful because it exploits sequences of symbols with repeats in it before the actual encoding process. This is why in JPEG compression we do a “zig-zag” scan on a section of quantized values before applying run length encoding on it to exploit the trailing zeroes resulting from the “zig-zag” scan.

Furthermore, discussing the compression ratios obtained in the table above, the reason why LZW compression ratio is below one is most likely because I am only using a maximum of 12 bits for the word length with the first initial 256 table entries set to all the possible “byte” values. Through some testing, I confirmed that my compression ratio was significantly worse when I increased the number of bits to use for the code length. Then as I reduced the number of bits the ratio got better. These results for the LZW only reflect the four sample wav files and so these findings I discovered are specific to them as well.

The Huffman encoding, I feel like it could have been better if I stored less data into the headers that I use as part of my decoding process. Although we are talking about a couple of bytes here but still I think it would have made the compression ratio better.

**Q. 2.**

**Report**

The techniques used in my lossy compression includes the following key techniques: Forward and Reverse DCT (discrete cosine transform), Quantization of Luminance and Chrominance values, and Huffman coding / decoding.

**DCT (discrete cosine transform):**

There were two ways to apply the DCT, but in my code I chose to go with the 8x8 DCT transformation matrix. I would then apply this DCT matrix using matrix multiplication on individual 8x8 blocks of the entire image. This gives us the DC and AC coefficients that we require for quantization explained below.

**Quantization of Luminance and Chrominance values:**

Similar to how I used a matrix for the DCT transformation, I also used matrices for both the Luminance and Chrominance. These two matrices were obtained from the JPEG specifications so it’s the same matrices used in JPEG. Just like it does in JPEG, I divide each 8x8 block of the Y channel 2D array with the luminance matrix values. Finally, I also divide the U and V channels 2D arrays with the chrominance matrix values.

**Huffman coding:**

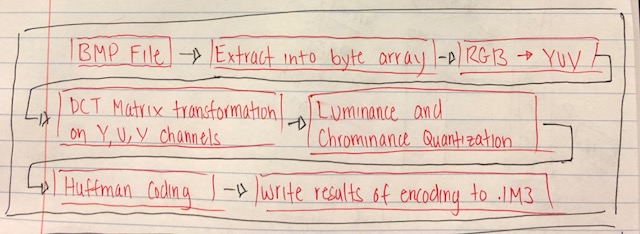
Huffman coding is used to encode the entire original header information of the BMP file and the quantized values obtained from quantization. Note that no “zig-zagging” and “run length encoding” is used because I didn’t have the time to implement that so the sequence that is encoded during Huffman coding is a sequential scan of the individual Y, U, and V quantized values stored in separate 2D arrays.

Furthermore, how the encoding process is programmed in my case is that I have a method to count the number of samples and how many times each sample appears. I store all these values into a HashMap where the “key” is the sample and the “value” is the number of times that sample appears. I then build the Huffman tree with the help of a priority queue (in Java the priority queue by default sorts it from smallest to biggest) where I always extract two items from the queue and insert a new node into the priority queue with the sum of the two extracted previous nodes pointers to those two. I keep doing this until there is only one item left inside the priority queue.

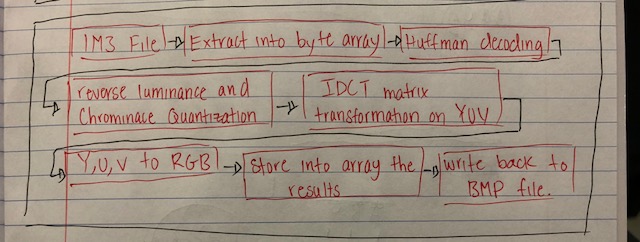
While building the tree I also create a HashMap at the same time where the “key” is the sample and the “value” is the codeword. This makes my life easier when I start encoding data because it’s easier to look at the sample’s corresponding codeword inside a HashMap than it is to traverse a tree programmatically in my own opinion.

Along with the encoded data, I also include extra information into the header such as the Huffman Tree structure and the data of each of its nodes. This way when I decompress the data I can first extract the Huffman Tree and with that I can reconstruct the encoded data back to its original state.

**Encoder**



**Decoder**



**Lossy Encoding / Decoding Speeds**

|  |  |  |
| --- | --- | --- |
| BMP File | Encoder (milliseconds) | Decoder (milliseconds) |
| P2-sample\_1 | 77 | 339 |
| P2-sample\_2 | 16 | 142 |
| Q3-sample\_1 | 42 | 30 |
| Q3-sample\_2 | 93 | 266 |

**Lossy Encoding Reconstruction Values and Compression Ratio**

|  |  |  |  |
| --- | --- | --- | --- |
| BMP File | Reconstruction Quality - PSNR | Compression Ratio | JPEG Compression Ratio |
| P2-sample\_1 | 29.322667685871192 | 6.307 | 24.874 |
| P2-sample\_2 | 30.57205505195419 | 5.823 | 24.216 |
| Q3-sample\_1 | 27.600019173844906 | 5.791 | 16.459 |
| Q3-sample\_2 | 27.56175509531438 | 5.937 | 20.127 |

The results of the compression ratios were obtained by first using the website below to convert the files to a JPEG files and then calculating the compression ratio by compressing their sizes.

<https://image.online-convert.com/convert-to-jpg>

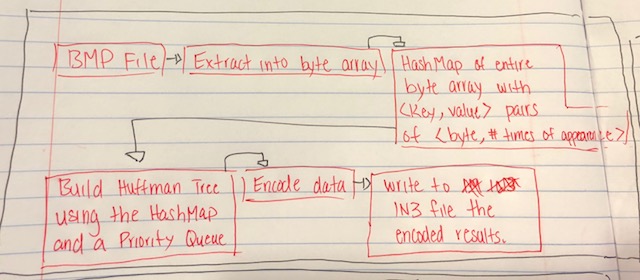
Some issues that are not really issues is that I did not implement “zig-zag” and “run length coding” which would have made the compression better. However, even without those two techniques I still achieved good compression results as shown above with each individual file’s compression ratio. Actually, I did run into some problems near the end trying to incorporate the “run length coding” and found out that it was harder than expected because when you encode the number of times a value is skipped you have to separate that value from the sample value or else you won’t know if what you’re looking at is one entire number or two separate numbers indicating the sample and skip value. A lot of code refactoring had to be done to get this to work which I attempted but failed.

**Q. 3.**

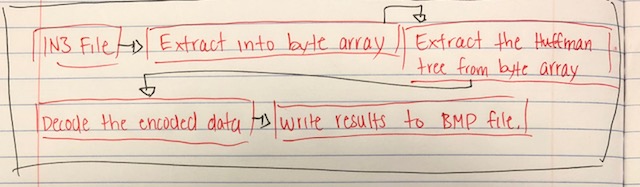
**Report**

The key techniques used the lossless compression is basically the standard steps of Huffman coding and decoding in lecture. Specifically speaking the key techniques used in my source code is as follows. Huffman tree built from a HashMap with the “keys” representing the values obtained from byte array containing the BMP file and the “values” as the number of times each of those values appear. Encoding the entire byte array using the Huffman tree and writing the results to a new file of type IN3. Note that I also store relevant information such as the Huffman tree and all its node data so that when I read in the IN3 file I would be able to decode the encoded sequence back to its original state.

**Encoder**



**Decoder**



**Lossless Encoding / Decoding Speeds**

|  |  |  |
| --- | --- | --- |
| BMP File | Encoder (milliseconds) | Decoder (milliseconds) |
| P2-sample\_1 | 105 | 5010 |
| P2-sample\_2 | 45 | 1313 |
| Q3-sample\_1 | 15 | 379 |
| Q3-sample\_2 | 53 | 4612 |

**Compression Ratio**

|  |  |  |
| --- | --- | --- |
| BMP File | Compression Ratio | TIFF Compression Ratio |
| P2-sample\_1 | 1.058 | 1.200 |
| P2-sample\_2 | 1.071 | 1.425 |
| Q3-sample\_1 | 1.044 | 1.766 |
| Q3-sample\_2 | 1.440 | 1.420 |

The results of the compression ratios were obtained by first using the website below to convert the files to a TIFF files and then calculating the compression ratio by compressing their sizes.

<https://image.online-convert.com/convert-to-tiff>

Some interesting issues I ran into when I was coding the Huffman code was that the compression rate wasn’t as good as I expected. Actually, it was really close to 1 for each file’s compression ratio, the reason why I think it is like this is actually because I was treating each individual byte as a symbol which means that in the worst case I have a most 256 total symbols which is the total number of values a byte can store [-128 to 127]. If I instead used two bytes instead of one byte when building the Huffman tree for encoding the compression ratio could have been better probably. The other reason could probably be because Huffman coding has the disadvantage that you have to store at least the Huffman tree for the decoding process, so this made the compression a little bit worse. I also ran into a terrible issue where I was doing String concatenation in my code when encoding and this made my program extremely slow to the point where it would take many minutes to compress and decompress. Luckily, I solved this issue by using Java’s StringBuilder method that uses buffers to store the string so “concatenation” is actually taking place which made my program much faster (extremely fast…).