bMatthew Lonis

Adeel Bhutta

CSCI-B 456

4 April 2018

Final Project Proposal

Algorithm/Topic: JPEG Image Compression

Group Members: Matthew Lonis

Description:

Step 1: The first step is to split the image into 8x8 non-overlapping blocks of pixels. If the image isn't evenly divisible by 8, we need to zero pad the image in order for it to be split into 8x8 non-overlapping blocks of pixels.

Step2: Transform the color space from RGB to YCbCr and then down sample the color space. We down sample the color space by taking advantage of the biological visual system of humans. Human visual systems have a lower acuity for color differences than for intensity differences which allows us to remove, or down sample, some of the color information in order to save space while also retaining minimal visual impact to the overall picture.

Step 3: Take the discrete cosine transformation (DCT) of each 8x8 block of pixels. This is useful for step 4 to filter out high frequency components in the 8x8 blocks of pixels. We can do this and have minimal impact to visual quality due to the human visual system's insensitivity to high frequencies. The DCT function converts pixels from the spatial domain into the frequency domain. The value at array index (0, 0) is a DC component and the rest ar AC components.

Step 4: In step 4 we perform quantization on each 8x8 DCT matrix. As stated in step 3, this allows us to remove high frequency components and only keep the lowest frequencies up to a certain point. This reduces the number of bits per sample.

Step 5: We perform a Zig-Zag scan on each 8x8 matrix to map it to a 1x64 array. This groups low frequency coefficients at the top of the array and high frequency coefficients at the bottom. This step is necessary for the following step.

Step 6: Perform differential pulse code modulation (DPCM) to each DC component of the arrays from step 5. DPCM encodes the difference between the current and previous 8x8 matrix.

Example

Original Arrays	DPCM on DC Component arrays
38 0 0 0 0 0 0 0	38 0 0 0 0 0 0 0

45 0 0 0 0 0 0 0	7 0 0 0 0 0 0 0
34 0 0 0 0 0 0 0	-11 0 0 0 0 0 0 0
29 0 0 0 0 0 0 0	-5 0 0 0 0 0 0 0

Step 7: Perform run length encoding on each array from step 5. A typical 1x64 array will have lots of zeros in it. This step encodes each set of zero as a (***skip value***) pair where skip is the number of zeros preceding a non-zero value in the quantized matrix and value is the actual coded value of the non-zero component. We can also group other non-zero values by this similar method shown in the examples below:

Example 1
Original Data Stream: 17 8 54 0 0 0 97 5 16 0 45 23 0 0 0 0 0 3 67 0 0 8 ⋯
Run-Length Encoded: 17 8 54 0 3 97 5 16 0 1 45 23 0 5 3 67 0 2 8 ⋯

Example 2
Original Data Stream: 5 5 8 9 9 9 9 9 9 30 30 30 22 22 22 22 22 22 12 12
Run-Length Encoded: 2 5 1 8 6 9 3 30 6 22 2 12

Step 8: We now need to perform Huffman coding on the DC and AC components to represent them by a smaller number of bits. We have to perform different codings for the DC and the AC components

DC Components
The DC coefficient is represented by a pair of binary symbols (Size, Amplitude) where Size is the number of bits needed to represent the coefficient and Amplitude contains the actual bits. We can code this using a pre-generated table found on slides 25 and 26 of the slides at https://www.slideshare.net/AishwaryaKM1/jpeg-image-compression-56894348.

AC Components
We encode the AC Components as a pair of $\left(\left(\frac{RunLength}{SIZE}\right), Value\right)$. RunLength is the length of consecutive values, SIZE is the number of bits needed to code the next nonzero AC component's value. The value is the actual value of the AC component. Both of these can be coded using pre-generated tables found on slide 28 of the slides at https://www.slideshare.net/AishwaryaKM1/jpeg-image-compression-56894348.

Next Steps
From here, I would need to save the appropriate information to a JPEG file. I also plan on writing a decoder which simply reads the JPEG file, decodes it, and displays the decompressed image. If I can't get saving to a file to work correctly I will instead implement a functional decode which is given the Image table along with the Huffman table for the image and decodes the image from there and returns it for display.

References
My references are as follows:

http://www4.ncsu.edu/~rhee/export/papers/TheJPEGStillPictureCompressionStandard.pdf

https://arxiv.org/ftp/arxiv/papers/1405/1405.6147.pdf

https://math.dartmouth.edu/archive/m56s14/public_html/proj/Marcus_proj.pdf

https://www.researchgate.net/publication/268523100_THE_JPEG_IMAGE_COMPRESSION_ALGORITHM

http://www.iraj.in/journal/journal_file/journal_pdf/1-10-139036859144-46.pdf

https://ac.els-cdn.com/S1077201485710054/1-s2.0-S1077201485710054-main.pdf?_tid=306bb1bb-db22-4c5f-b88c-bda38f187ed9&acdnat=1522776649_26a516e5427579b7267d3792cbe6c676

https://www.slideshare.net/AishwaryaKM1/jpeg-image-compression-56894348