Comp311 Assignment 1 Report

Byron Miles 220057347

Usage

To Compile: make

The programs are written in C++. I have included a makefile, so to compile the programs (encode and decode) simply navigate to the appropriate directory and type 'make'.

You can also type 'make encode' and 'make decode' to compile each separately, however they both make use of the the same classes / object code, so the second one to be complied is mostly just linkage.

I have also included a 'clean' directive in the makefile (i.e. type 'make clean') that will delete the object files (*.o) as well as the 'encode' and 'decode' executables in case a full recompilation is required.

To Run: encode a.ppm a.uneif / decode a.uneif a2.ppm

The encoding program is called 'encode' and the decoding program is called 'decode'. They both take two arguments, a source file and a destination file.

In the case of encode, the source file must a PPM file of type 'P6', with a maximum RGB value of 255. An example of encoding a file would be to type 'encode baboon.ppm baboon.uneif'.

In the case of decode, the source file must be a UNEIF file. An example of decoding a file would be to type 'decode baboon.uneif baboon2.ppm'.

Note: The extension of the file names don't really matter, as the neither encode nor decode makes any checks for / of the file extension. The specified files must be of the correct format however.

PPM files to use: must be 'P6' with max rgb of 255

As I said before the PPM file must of type 'P6' with a maximum RGB value of 255, i.e. 8 bits per colour, 24 bits per pixel. While you should be able to use any PPM file with that format, I have tested it, and it does work, with the .ppm files from http://www.hlevkin.com/TestImages/classic.htm .

Note: While I mostly used baboon.ppm as the test image while writing the program I did not make any assumptions based on the specifics of baboon.ppm.

The encoding process

Reading the PPM Image

For simplicity I chose to restrict the types of images that can be encoded to the 'P6' type of PPM Image with a maximum RGB value of 255, that is, each colour value is stored in 8 bits (a char), with each pixel consisting of 3 colours, for a total of 24 bits per pixel.

In reading the PPM Image I first read the header information, and while I made an effort to support most header comment arrangements, it may not be fool proof. I also rely on the PPM format specification that there be a single white-space character between the maxrgb value and the data.

To read the data I simply extracted it one char (8 bits) at a time, putting the char into red, green and blue vectors in turn. This makes the data one dimensional, and as such easier to deal with, it does have the disadvantage of not allowing for optimal compression, however I was more concerned with just making it work than making it optimal.

Transform coding : One dimensional Haar wavelet transform

Once I had the three colour vectors (red, green and blue) I converted the values to floats and performed a one dimensional Haar wavelet transform one each vector.

To prevent complications due to odd length vectors at any level of the transform I chose to break the vectors into blocks, with each block being SIZE = 2 to the power of X, with SIZE being less than or equal to a pre-defined value. I then performed X levels of Haar wavelet transformation of each block.

Quantisation : Uniform scalar quantisation

For the quantisation stage I used a very simple uniform scalar quantisation process.

I basically just took each value, added half of a pre-defined range, in this case 512 (-255 to 255), in order to make each value positive, added +0.5 so the numbers would be 'rounded' to the nearest whole, divided by a pre-defined step size (e.g. a step size of 4 would yield values 0, 4, 8, 12, 16, … on the decoding side) and truncated the result.

Entropy coding : Huffman coding

For the entropy coding stage I used Huffman coding, however due to the trickiness of implementing it, I regret to say that while my implementation works correctly, it is a little convoluted and not as efficient as it could be time wise.

In order to implement the Huffman coding I first started with a Node (HuffmanTree.cpp), which holds values for the symbol, count and links to any child nodes. I use these Nodes to build the Huffman coding tree.

I then created a HuffmanTree class (HuffmanTree.cpp), that takes a vector of symbols and a vector of their corresponding counts and uses that to build a binary tree as per the Huffman algorithm. This class is also used to build a code table (a vector of symbols and a vector of their corresponding codes) from the tree.

I then encapsulated the HuffmanTree within a Huffman (Huffman.cpp) class. The Huffman class takes source vectors and generates a vector of the unique symbols within it and their corresponding counts. The reason for this was to allow the merger of the three colour

vectors (red, green and blue) into a single vector of unique symbols.

The final (merged) vector of symbols and counts can then be passed to, and used by, the Huffman class (via the HuffmanTree class) to create, and hold, a table of symbols and their corresponding codes. This table is sorted by code length to facilitate faster look up of symbols from codes.

Once all this is done, the Huffman class can then be used to relatively easily translate between symbols and codes.

Also note that in my implementation the entropy coding isn't really a discrete step (like the transform coding or quantisation), but is tied in with the reading and writing of the compressed (.uneif) file in the UNEIF class (UNEIF.cpp).

Writing the UNEIF file

The uneif format is fairly simple and consists of a header section formatted as follows:

WIDTH HEIGHT

NUMBER_OF_SYMBOLS

LIST_OF_SYMBOLS (E.G. ## ## ## ## ##)

LIST_OF_COUNTS (E.G. ####### ##### ### ## #) A_SINGLE_WHITE-SPACE_CHAR

Following the single white-space character after the list of counts is the coded data.

The reason for this format is that I needed to retain the width and height of the original image as well as a count of the total number of symbols that were encoded and written (i.e. width * height). I also needed to know how many items were in the list of symbols, what they were, and their corresponding counts so that I could recreate the Huffman tree on the decoding side.

To write the data I used the Huffman class to translate each symbol into it's code, and pushed that code onto a buffer, once I had 8 'bits' (in my implementation, boolean values in a queue) on the buffer, I used an STL std::bitset<8> to create an 8 bit number that I then turned into a char and wrote to the file. The data is written in pixel order, that is, a code for a red value, then the code for a green value, then the code for a blue value and so on till the end of the file.

Note: As there was no guarantee that the final stream of coded values would be evenly divisible by 8, for the final char (i.e. any bits left in the buffer after coding all the symbols) I simply created the last char from what I had and left in the buffer with rest as 0's. This is one of the reasons I needed the width and height values in the header.

The decoding process

Reading the UNEIF file

To read the compressed image file I first read the header information, as the header format in simple, without any comments or extra white-space, this is fairly straight forward.

Then using the list of symbols and counts from the header I used the Huffman class to recreate the coding table that was used to encode the file.

Using the coding table, and taking advantage of the unique prefix quality of Huffman coding I read a char (8 bits) from the data, used the char to create an STL std::bitset<8> and pushed the bits (again boolean values in a queue) on to a buffer.

Once this buffer was full enough I then popped each 'bit' off the buffer in turn, pushing it to a code buffer (in this case a vector of char's). I then passed this code buffer to the Huffman object to see if it corresponded to a symbol. If it did I took the symbol and pushed it to each colour vector (red, green or blue) in turn, wiped the code buffer and incremented the count of symbols that had been decoded. If it didn't correspond to a symbol I simple popped the next bit off the buffer, pushed it to the code buffer and tried again.

I repeated this process (filling the bit buffer, using it to fill the code buffer, and pushing symbols onto the colour vectors) until I had decoded and pushed all of the symbols (i.e. count = width * height).

Note: This is far from an optimal implementation from a time perspective, but again I was more focused on just making it work, and doing it like this allowed me to use a similar structure and method of code to symbol translation as the encoding side.

De-quantisation

The de-quantisation stage is pretty much just the reverse of the encoding side in that I first multiply the value by the step size, then subtract half the pre-defined range to re-centre the values around 0.

De-transformation

Again this is pretty much just the opposite of the encoding side, and due to the fact that each transformed block is 2 to the power of X in size, it meant that I didn't have to deal with any weird 'edge' values, making the process of reversing the one dimensional Haar wavelet transform quite straight forward.

Error correction

Despite, or more probably because of, the simplicity of the quantisation and transformation stages, the combination of the two induced 'out of range' errors in the data set. That is, after the de-transformation stage the values in the red, green and blue vectors didn't necessarily fall into the initial range of 0 to 255. This has the effect of producing 'spots' on the image, the severity of which seems to be linked to the step size of the quantisation.

To fix these errors I simply set all values below 0 or above 255 to 0 and 255 respectively.

Note: If you would like to see the effects of these errors for yourself you can turned off the error correction by changing the 'correctErrors' variable near the top of the 'Decode.cpp' file to false. See the comments section below for more details.

Writing the PPM Image

With the data returned to some semblance of it's original self, I then passed it, along with the appropriate header information, including the width and height values I stored in the uneif file header, to the PPM class.

I kept the PPM header fairly simple, and wrote it as follows:

P6

Created by PPM.cpp

WIDTH HEIGHT

MAX_RGB A_SINGLE_WHITE-SPACE_CHAR

After the header I simply iterated through the colour vectors, writing each pixel (red, green, blue) in turn.

Comments

Variable customisation

While not exactly user friendly, you can customise some of the variables that the various stages of encode and decode use. To do this:

- Open both the Encode.cpp and Decode.cpp files.
- On about line 22 of each file you can change the 'quantiseStep' variable.
  - This controls the step size used in quantisation, the higher it is the lower the number of unique symbols that will need to be given a code, and thus the smaller the .uneif file, but the worse the degradation of the image at the other end.
- On about line 23 of each file you can change the 'transformBlock' variable.
  - This controls the maximum size of the blocks used in the transform coding stage. This must be a power of 2 (e.g. 128, 256, 512, 1024) but you can make it as big as you like (within the limits of an unsigned int). Note: not all blocks will be this size, just as many as possible.

Please ensure both variables have the same values in both files!

Note: Images encoded with one set of values will not decode properly with another set of values.

- On about line 25 of Decode.cpp there is an additional variable called 'correctErrors'
  - Changing this to false will turn of the error correction code that prevents 'spots' from appearing on the image.

Note: A recompilation of the programs will be required after each change.

Compression ratio

For comparison, without significant image degradation my compression scheme can generally reduce the file size to about half; JPEG can easily reduce it about one tenth the PPM size.

Clearly this implementation of image compression isn't going to be winning any awards, and while I probably won't be taking up codec writing as a career it did give me a good understanding of the principals involved. Also, the way it is implemented it shouldn't be too difficult to change / improve some of the steps in order to get better compression ratios; unfortunately I don't have the time to do that at this stage.

Performance

As I mentioned, my implementation is not the most efficient time wise. However now that I have a better idea of how it all works and fits together I can think of a number of improvements that could be made.

It works in it's current state though, and as performance is not stated as a key criteria for the assignment, I am hesitate to risk breaking it.

Runtime errors

I should also note that while I made an effort to test and provide error checking within the code itself, it my not be fool proof and may not be able to handle all PPM files equally well. As I mentioned before I have tested it with the PPM files from http://www.hlevkin.com/TestImages/classic.htm , so if your PPM file isn't working, try some of those.