

Supplement to Computer Science Chapter 3 – Digital Image Processing

[Programming Exercise – Digital Imaging > Huffman Encoding](#)¹

Introduction:

Huffman encoding is a compression algorithm that can be applied to both text and image files. It achieves a good compression rate by assigning shorter codes to values that appear more frequently in a file and longer codes to those that appear less frequently. Huffman encoding is lossless; that is, no information is lost in the encoding, so that the decoded file is identical to the original file.

Huffman encoding is used as one of the last steps in JPEG and MPEG compression. Variants of Huffman encoding are also used in commercial compression algorithms such as PKZIP and Stuffit.

An interesting property of Huffman encoding is that it allows for variable-length codes. That is, the code for one color in an image file may be six bits while another may be five bits. This may seem odd. Imagine a string of bits representing a Huffman encoded image file. You want to decode the file, but if the code for one pixel in the file is not necessarily the same number of bits as the code for the next, how do you know where the value for one pixel ends and the next one begins? The trick is that, because of the way the codes are created, you can be sure that no code is a prefix of any other.

The steps in Huffman encoding are as given below. We assume that we're working with grayscale image files. Extending the algorithm to RGB color files is straightforward.

1. Count the frequency of occurrence of each of the 256 colors in the file.
2. Create a Huffman tree working from the leaf nodes up to the root as follows:

Create a node for each color in the file. Each node has stored with it a color from the image file and the frequency of that color in the image. These nodes will be the leaf nodes of a Huffman tree, which is constructed from the leaf nodes up as follows:

Repeat until you have created a root node that joins all existing nodes in one tree

Among all existing nodes that do not yet have a parent node, find the two nodes with the lowest frequency. In case of ties, make an

¹This material is based on work supported by the National Science Foundation under Grant No. DUE-0127280. This programming assignment was written by Dr. Jennifer Burg (burg@wfu.edu).

arbitrary choice. (The basis for the choice must be applied consistently, and the decoder must use the same basis for breaking ties.) Make a parent node for the two selected nodes. For the parent node, the frequency stored with the node is equal to the sum of the frequencies for the two children nodes.

3. For each node in the tree other than the leaf nodes, mark the arc to the left child of the node with a 0, and the arc to the right child the node with a 1.
4. For each leaf node (which represents a color in the image file), assign a code that corresponds to the numbers on the arcs that must be traversed to get to the leaf node from the root.

The four steps above assign a code to each color. Note that not all codes are the same length, and no code is a prefix of any other.

The image file can be encoded with these codes by substituting each pixel value (i.e., color) with its code.

Instructions:

The Assignment

Using the programming language of your choice, implement the Huffman encoding algorithm.

Your program should take as input a raw grayscale image. Prompt the user for the name of the input file. The image file should have the width and height of the image as integer values at the head of the file.

For convenience in verifying correctness, the program should ask the user if he or she wants the frequency table and code table output to the screen. Prompt the user for the name of the encoded image file, and output this file to disk.

Output to the screen the compression ratio for the compressed image.

Put the actions above in a loop in your main function, asking the user if he or she wants to compress another image.

Before Writing the Program

To run your program, you will need some raw grayscale image files as input, so create these first or ask your instructor if he or she plans to create them for you. The image files should have one byte per pixel and a header of two integers giving the width and height of the image. The input image files can be created by saving an image (.bmp, .jpg, etc.) in .raw format in a standard image processing program.

Tip for Creating Input Images



If you can't figure out how to put the width and height on the raw image file as a header, save the image file in .raw format with no header, but make a note of the width and height as you do so. Then write a simple program that reads in the raw file and creates a new file by writing the width and height out first and then the remainder of the image data. Having the width and height on the file as a header saves the user the trouble of typing the width and height in each time an image is processed.

Two Possibilities for Implementation – Simulation or the Real Thing

If you can't do bit-level operations in your chosen programming language, then you'll need to simulate the Huffman encoding algorithm. That is, you won't actually reduce the eight-bit pixel values to variable bit-length codes and thus reduce the file size. Instead, create an encoding table from the Huffman tree where the codes in the tables are stored as strings of characters containing only 0's and 1's. You can compute the compression ratio based on the lengths of the strings in the code table and the number of occurrences of each color in the file.

If you are able to do bit-level operations, you can do a more realistic Huffman encoding, using bit packing operations to reduce the file size.

Tips for C/C++ Programmers



You can use *unsigned chars* for your pixel data. An eight-bit *unsigned char* is just the right size to store a grayscale pixel value. Instead of using the `>>` file-input operator to read your pixel data, use the *read* function associated with random access files. The `>>` operator skips non-printable characters, which will cause you problems if one of your pixel values happens to be the ascii code for a non-printable character. You can use the *read* function to read exactly one byte at a time from the original image file.



You can take advantage of the fact that *chars* can be treated like integers in C/C++. For the frequency table, you can create an array of 256 *unsigned chars*, read each pixel value into a single *unsigned char* variable, and use that variable as the index into the array.

Ideas for Further Experimentation and Analysis

- Implement the decoding algorithm. Compare the original file with the file you get after encoding and decoding. Verify that Huffman encoding is lossless.
- Implement Huffman encoding on an RGB color file.

- Try Huffman encoding on different types of image file. Is it more effective on certain types of images? If so, how would you characterize the images it works best on, and why?
- Read variations of Huffman encoding (e.g. adaptive Huffman encoding), and consider how your implementation could be improved.