CS302 Data Structures

Spring 2012 – Dr. George Bebis

Programming Assignment 4

Due Date: 4/17/2012

Goal: In today's world, many images are transmitted across the internet. To send them efficiently as well as to store them efficiently requires that we compress them before we send them and then uncompress them when they are received. In this assignment, you will be using a priority queue and binary trees to implement an image compression/uncompression algorithm called *Huffman coding*. Specific objectives include:

- Use a priority queue in image compression
- Implement and manipulate binary trees
- Learn about Huffman Coding
- Learn about bit and byte input/output

Huffman coding: Each pixel in a PGM image is represented by one byte or 8 bits. Huffman coding compresses an image by encoding the pixel values so that not all pixel values require 8 bits of storage. The key idea is assigning fewer bits to pixel values that appear more frequently in the image and more bits to pixel values that appear less frequently in the image. As a result, the size of the compressed image is smaller than the original one. Let us consider the 6x6 image shown below where every pixel is encoded using 8 bits. Storing this image would require 36x8= 288 bits. Since pixel values are between 0 and 7, we could assume 3 bits per pixel only; in this case, storing the image would require 36x3=108 bits. Using Huffman coding, however, it would require 93 bits.

1	3	1	5	5	7
5	5	5	1	2	1
6	6	4	0	6	2
6	4	6	6	6	6
3	3	5	5	5	5
6	6	5	5	7	5

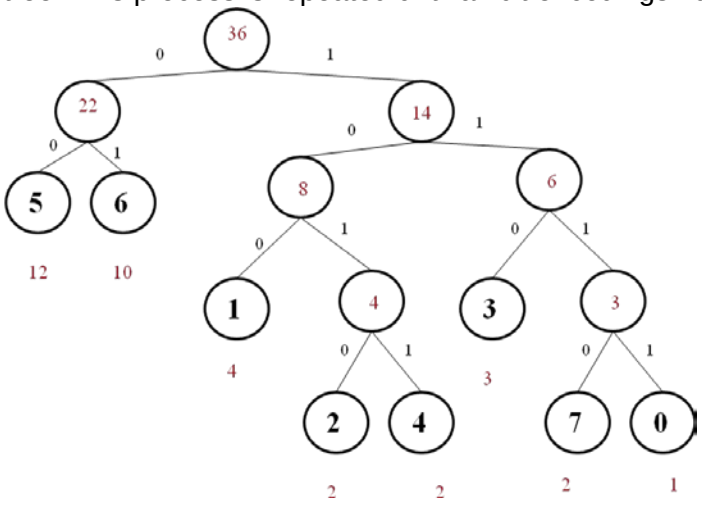
f(0) =1
f(1) =4
f(2)=2
f(3)=3
f(4)=2
f(5)=12
f(6)=10
f(7)=2

Figure 1. A simple 6x6 image, its pixel frequencies, and its histogram.

To compress an image, you need a ***table of bit encodings***, (e.g., an ASCII table, or a table giving a sequence of bits that's used to encode each pixel value). This table is constructed from a ***coding tree*** using root-to-leaf paths to generate the bit sequence that encodes each pixel value. Figure 2 shows an example of a Huffman coding tree and the corresponding table of bit encodings.

Generation of Table of Bit Encodings: In a coding tree, pixel values are stored at the leaves of the tree. Also, a left-edge is labeled 0 (or 1) and a right-edge is labeled 1 (or 0). The Huffman code for any pixel value can be obtained by following the root-to-leaf path (i.e., leaf corresponding to the pixel value) and concatenating the 0's and 1's in the path. Figure 2 shows an example of a table of bit encodings (right) generated by a coding tree (left).

Encoding/Decoding: To encode an image, we replace each pixel value by its corresponding bit encoding. Using the encoding shown in Figure 2, for example, the first row of the image shown in Figure 1 is encoded as follows: 10011010000001110. To decode a stream of pixel encodings, start at the root of the encoding tree, and follow a left-branch for a 0, a right branch for a 1. When you reach a leaf, write the pixel value stored at the leaf, and start again at the top of the tree. This process is repeated until all bit encodings have been decoded.



0	**1111**
1	**100**
2	**1010**
3	**110**
4	**1011**
5	**00**
6	**01**
7	**1110**

Figure 2. (Left) Huffman coding tree corresponding to the image shown in Figure 1; (Right) Coding of pixel values. Note that the code length of each pixel is inversely proportional to its frequency.

Generation of Huffman Coding Tree: We now describe the algorithm for constructing an optimal coding tree. Let's assume that each pixel value has an associated weight equal to the number of times the pixel value occurs in an image. In Figure 1, for example, pixel value 0 has a weight 1 while pixel value 5 has a weight 12. When compressing an image we'll need to calculate these weights. Huffman's algorithm assumes that we're building a single tree from a group (or forest) of trees. Initially, all the trees have a single node with a pixel value and the character's weight. Trees are combined by picking two trees, and making a new tree from the two trees. This decreases the number of trees by one at each step since two trees are combined into one tree. The algorithm is as follows:

1. Begin with a forest of trees. All trees are one node, with the weight of the tree equal to the weight of the pixel value in the node. Pixel values that occur most frequently have the highest weights. Pixel values that occur least frequently have the smallest weights.

2. Repeat this step until there is only one tree: Choose two trees with the smallest weights, call these trees T1 and T2. Create a new tree whose root has a weight equal to the sum of the weights T1 + T2 and whose left subtree is T1 and whose right subtree is T2.

3. The single tree left after the previous step is an optimal encoding tree.

The tree shown in Figure 2 is an optimal tree for encoding the image shown in Figure 1; that is, there are no other trees with the same pixel values that use fewer bits to encode the image in Figure 1 There are other trees that use 92 bits; for example you can simply swap any sibling nodes and get a different encoding that uses the same number of bits.

Implementation: Your program will read an image and compress it using your Huffman coding implementation. The compressed image will be written to a file. That compressed file will then be read back by your program and uncompressed. The uncompressed image will then be written to a third file. Since Huffman coding is a “lossless” compression technique, the uncompressed image should match the original image exactly. You should add an extra option in driver.cpp, for compressing/uncompressing an image. This is a summary of the main steps to be executed upon choosing this option:

1. Read the specified image and count the frequency of all pixels in the image.
2. Create the Huffman coding tree using a Priority Queue based on the pixel frequencies.
3. Create the table of encodings for each pixel from the Huffman coding tree.
4. Encode the image and output the encoded/compressed image.
5. Read the encoded/compressed file you just created, decode it and output the decoded image.

Your program must provide the following required output: (i) the pixel frequencies (i.e., image histogram; can be visualized as a gray-scale image) (ii) the table of pixel values and their bit encoding (sorted from smallest to largest) (iii) original image size and compressed image size and (vi) compressed image and uncompressed image.

Comments

- The Huffman coding tree is based on a binary tree. You may design and implement your own "Huffman Tree" class from scratch, but the Binary Tree class discussed in the lectures might make a good starting point.

- Note that two passes of the original image are required. The first pass counts the frequency of each pixel while second pass performs the actual encoding.

- The compressed file should store (i) the number of pixels, (ii) the coding tree and (iii) the stream of bits. Store the number of pixels and coding tree in ASCII format. The tree can be stored at the beginning of the file (i.e., header). One using a pre-order traversal. You must differentiate leaf nodes from non-leaf nodes. One way to do this is write a single bit for each node, say 1 for leaf and 0 for non-leaf. For leaf nodes, you will also need to write the pixel value stored. For non-leaf nodes there's no information that needs to be written, just the bit that indicates there's a non-leaf node.

- Store the stream of bits in “chunks” of 8 bits or 1 byte at a time. This can be done by”pushing” 8 bits at a time in an “unsigned character” using bitwise operators (e.g., logical OR); then use “write” to save the unsigned character in the file (see WriteImage.cpp). When reading the stream of bits from the compressed file read them in “chunks” of 8 bits or 1 byte at a time into an unsigned character using “read” (see readImage.cpp). Then, extract the

bits using bitwise operators. Alternatively, you can use C++ classes to read and write bit streams; here is an example:

http://www.codeproject.com/Articles/332219/C-Classes-to-Read-and-Write-Bit-Stream

Instructions

Each function should be discussed in a separate section with the name of the section being the same as the name of the function. Functions that you have implemented in previous assignments do not need to be described again (only if you have made significant modifications). The sections should be clearly separated from each other.