University of Hawaii EE 367 Machine Problem 3 Huffman Code

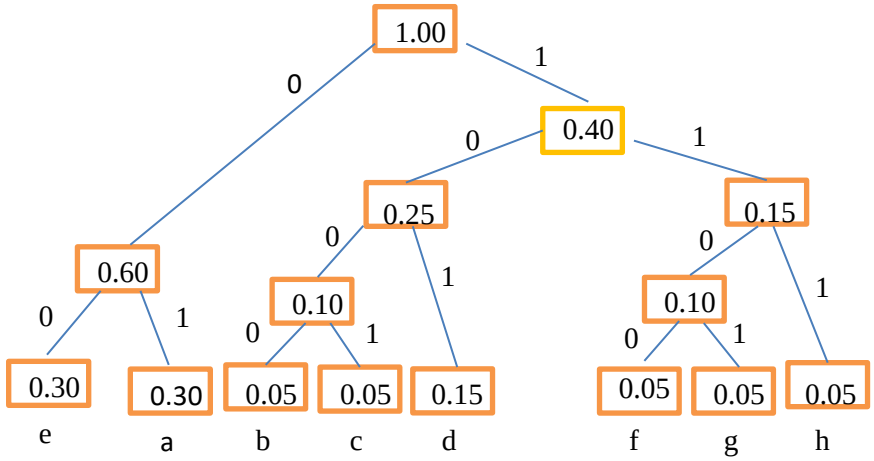
Last update: April 6, 2018

1. Introduction

A Huffman code is a data compression code used to convert data files (or streams of data) into a smaller file. The coded file can be stored, then later decompressed back into the original file. The following is an example, which is a table of data (ASCII chars) and their frequencies.

Symbol	Frequency
a	0.30
b	0.05
c	0.05
d	0.15
e	0.30
f	0.05
g	0.05
h	0.05

The following is a corresponding Huffman tree (trie), and a prefix code.



Symbol	Frequency	Code Word
a	0.30	01
b	0.05	1000
c	0.05	1001
d	0.15	101
e	0.30	00
f	0.05	1100
g	0.05	1101
h	0.05	111

2. Encoding

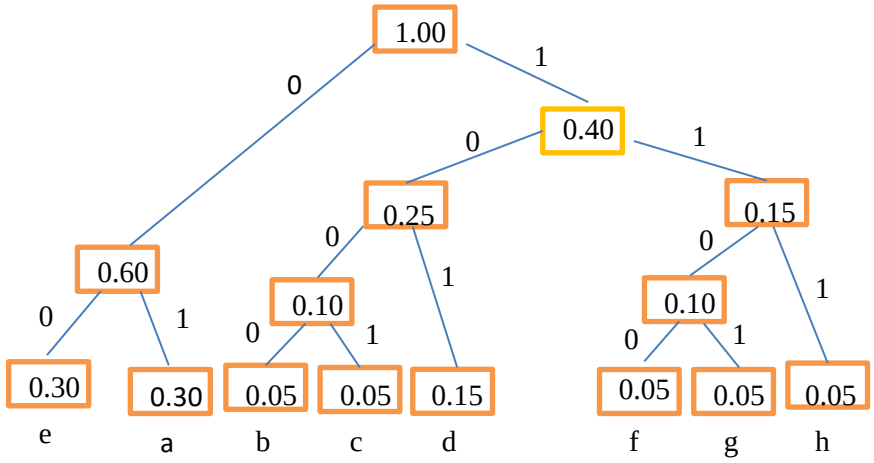
Here's how to compress a file

- Step 1: Scan the input data file and collect statistics, i.e., the frequency of each ASCII value. The frequency can be the actual number of occurrences rather than a fraction or probability.
- Step 2: Create a Huffman trie. From the trie, create a codebook
- Step 3: Create an encoding of the Huffman trie using pre-order traversal (this will be explained below).
- Step 4: Rescan the input data file and convert each byte into a codeword to create an encoding of the input data.
- Step 5: Create an output file which contains the encoded Huffman trie and the encoded data.

The following is a way to encode a Huffman tree (trie). Note that such a tree has leaf nodes representing data bytes, while internal (nonleaf) nodes do not represent data bytes. The encoding of the tree is accomplished by doing a tree traversal and using pre-order to record the nodes that are visited. Each internal node is recorded by the bit '1', while a leaf is recorded by the bit '0' followed by the data byte (8-bits) of the node. For example, to record the leaf node for 'a', we have '0 01100001', where the first bit indicates a leaf and the next 8 bits "011000001" is the ASCII representation of 'a'. As another example, the following tree can be encoded as

[1] [1] [0, "e"] [0, "a"] [1] [1] [1] [0, "b"] [0, "c"] [0, "d"] [1] [1] [0, "f"] [0, "g"] [0, "h"],

where the symbols are shown (e.g., "e") rather than their 8-bit binary representation.



The encoding of the tree is

[1] [1] [0, "e"] [0, "a"] [1] [1] [1] [0, "b"] [0, "c"] [0, "d"] [1] [1] [0, "f"] [0, "g"] [0, "h"],

Now let's calculate the maximum number of bits to encode the Huffman tree. The maximum number of data bytes is 256. Each data byte is a leaf. From what we know of binary trees, the number of internal nodes is the number of leaves minus 1. The number of bits per leaf node is 9, and the number of bits per internal node is 1. Thus, the maximum total number of bits is

= 256 x 9 bits + 255 x 1 bit = 256 x 9 bits + 256 x 1 bit – 1 bit = 256 x 10 bits – 1 bit = 2560 – 1 bits = 2559 bits.

Since the encoded Huffman tree has length less than 2560, we can represent the length with a binary number of length 12 bits.

The encoded file will be the concatenation of the following

- Length of the encoded Huffman trie, which is a 12-bit binary number
- Encoded Huffman tree, which is less than 2560 bits
- Encoded data file

3. Decoding

The decoder will input the encoded file and do the following:

- Step 1: Read the first 12 bits and convert it into an int value n.
- Step 2: Read the next n bits, which is the encoded Huffman trie, and recreate the trie.
- Step 3: Decode the rest of the file using the Huffman trie, and write that into an output file, which will become the decompressed data file, i.e., the original file

4. Tasks

We will now cover the tasks, and then the submission instructions and grading policy. There are five tasks:

- Task 1. Find the file 'encode367.c'. It will
    - It's use is './encode367 <input data file> <compressed data file>'
    - It scans the input data file and determines the frequency of the bytes in the file
    - It builds a Huffman trie using the function 'build_huffman_trie'
        - Note: the data bytes of the input file are in the leaves of the trie
        - Note: data bytes that are not in the input file are not included in the trie
    - It computes the codewords for the leaves and stores them in the leaves.
    - It displays
        - The size of the encoded Huffman trie
        - The size of the encoded data in the data file
        - The size of the compressed file
    - This program doesn't work
        - Function 'build_huffman_trie' doesn't build a Huffman code trie
            - It does build a trie, but not a Huffman trie
            - You will fix this in this task
        - The program doesn't output anything.
            - You will fix this in the next task, Task 2
    - Your task is to
        - Rewrite function 'build_huffman_trie' so it works
            - You can change everything in it
        - Hint: When building a Huffman trie, you can use a brute-force approach to find nodes with minimum frequencies rather than using heaps.

        - Note: There is a function 'debug_display_trie'. You can delete any calls to this function, but do not delete the function itself. I may use it when grading.
- Task 2: This is a continuation of Task 1
    - Have encode367.c output a compressed file which has the following format:
        - First 12 bits is n, the number of bits in the encoded Huffman trie
        - Next n bits is an encoded Huffman trie using preorder
            - 1: means a nonleaf (internal) node
            - 0: means a leaf, which is followed by an 8-bit binary representation of the leaf's data byte.
        - The rest of the bits are the Huffman encoded data from the input file
        - The output file should store characters '0's and '1's rather than bits.
            - This will make it easier to debug since you can view the output
            - Of course, your output will be much larger, by a factor of 8
- Task 3. Write a decompression program 'decode367.c' that has as input compressed files from Task 2 and outputs the decompressed (original) file
    - Usage: ./decode367 <input compressed file> <output decompressed file>
- Task 4. You will use your encode367.c to compress the files:
    - TheLottery.html
    - SomeSpanishStoryIDontKnow.html
    - FrenchLanguage.html

  Determine the *data compression ratio* of the each of the three files:

  The compression ratio = (size of compressed data file)/(size of uncompressed data file)

  Also compress the files using gzip and determine the compression ratios. (Note: 'ls –l' will list files in a directory along with their sizes in bytes.)

  Which compression algorithm is better, Huffman coding or gzip?

  In your README file, write your compression ratios for each file. You should have two ratios per file, for encode367 and gzip.

Submission Instructions and Grading:
You must complete each Task in order. The maximum number of points is 30 points. Grading:
None of the tasks: 15
Task 1: 19 pts
Tasks 1-2: 23 pts
Tasks 1-3: 27 pts
Tasks 1-4: 30 pts

Submit
- Working program files encode367.c and decode367.c
- README file, which includes your calculations for Task 4, i.e., the compression ratios for your Huffman code and gzip, as well as your analysis of which compression algorithm is better.