Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.087: Practical Programming in C

IAP 2010

Lab 2: Data compression

In-Lab: Wednesday, January 20, 2010 **Due:** Monday, January 25, 2010

Overview

Assume that the data consists of stream of symbols from a finite alphabet. Compression algorithms encode this data such that it can be transmitted/stored with minimum size. Huffman coding is a *lossless* data compression algorithm developed by David A. Huffman while he was a PhD student at MIT. It forms the basis of the 'zip' file format. It has several interesting properties:

- It is a variable length code. Frequent symbols are encoded using shorter codes.
- It is a prefix code. No code occurs as a prefix of another code. This property is also called as instant decoding. The code can be decoded using current and past inputs only.
- It is the best possible prefix code (on an average)–it produces the shortest prefix code for a given symbol frequency.

Generating code

Assume that we know the frequency of occurence of each symbol. We also assume that the symbols are sorted according to the frequency of their occurence (lowest first). The procedure to generate the code involves constructing a tree that proceeds as follows

- Intially all symbols are leaf nodes.
- The pair of symbols with the smallest frequency are joined to form a composite symbol whose frequency of occurence is sum of their individual frequencies. This forms the parent node in the binary tree with the original pair as its children. This new node is now treated as a new symbol.
- The symbols are re-arranged according to the new frequency and the procedure is repeated until there is a single root node corresponding to the composition of all the original symbols.

. After N-1 (N being the number of symbols) iterations the tree building is complete. Now, each branch is labelled with '1' (if right) or '0' (if left). The code for each symbol is the string of 1s and 0s formed when traversing the tree from the root to the leaf node containing the symbol.

Consider the following example: Let {('a',0.01),('b',0.04),('c',0.05),('d',0.11),('e',0.19),('f',0.20),('g',0.4)} be the exhaustive set of symbols and their corresponding frequency of occurence. We can represent the code formation through the following table and the corresponding tree.

iteration	tree
1	{('a',0.01),('b',0.04),('c',0.05),('d',0.11),('e',0.19),('f',0.20),('g',0.4)}
2	{('ab',0.05),('c',0.05),('d',0.11),('e',0.19),('f',0.20),('g',0.4)}
3	{('abc',0.1),('d',0.11),('e',0.19),('f',0.20),('g',0.4)}
4	{('e',0.19),('f',0.20),('abcd',0.21),('g',0.4)}
5	{('abcd',0.21),('ef',0.39),('g',0.4)}
6	{('g',0.4),('abcdef',0.6)}
7	{('abcdefg',1.00)}

Table 1: The table illustrates the formation of the tree from bottom-up

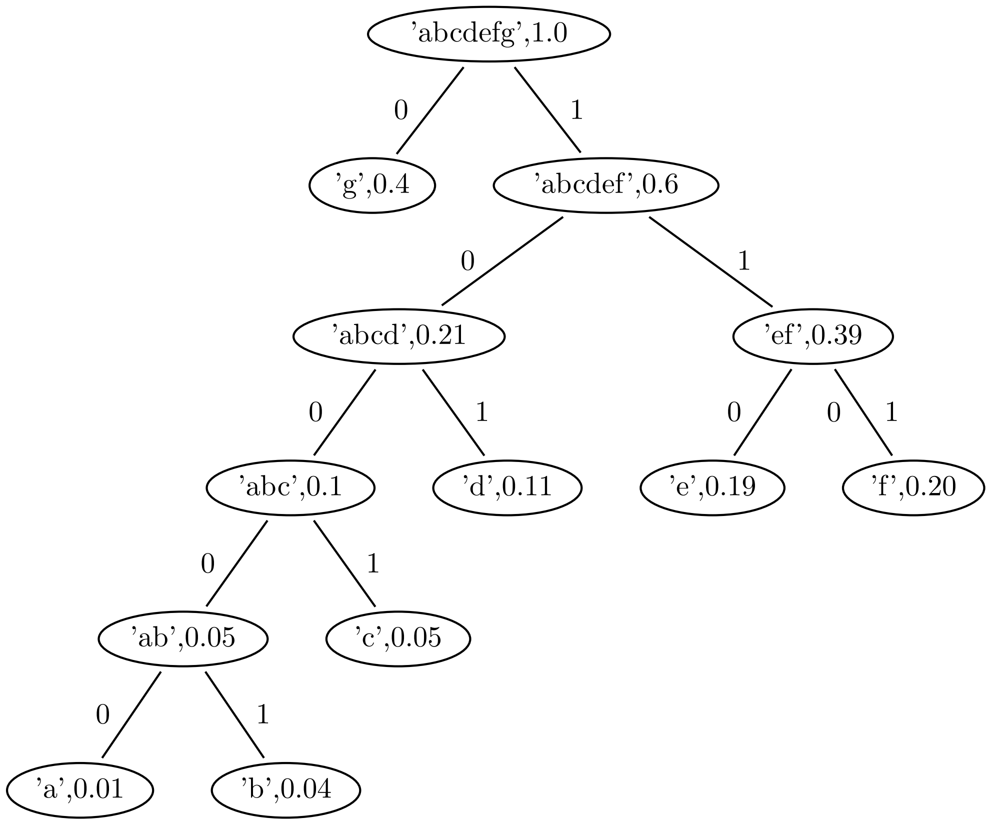

Figure 1: Huffman code tree corresponding to the given symbol frequencies

Part A: Implementing a Huffman decoder (In lab)

Instructions

(a) Please copy the sample code (decode.c) from the locker. ('/mit/6.087/Lab2/decode.c')

(b) Please go through the code to understand the overall structure and how it implements the algorithm.

Things to do:

(a) The symbol tree has to be recreated given the mapping between symbols and its binary string (see code.txt). The function `build_tree()` implements this functionality. Please fill in missing code.

(b) Given the encoded string and symbol tree, write the missing code to generate the decoded output. Hint: for each string, traverse the tree and output a symbol only when you encounter a leaf node.

(c) Finally fill in the missing code(if any) to free all resources and memory before exiting the code.

Output The program outputs a file 'decoded.txt' containing the decoded output. The output should be "abbacafebad".

Part B: Implementing a Huffman encoder (Project)

In this part, we will implement a Huffman encoder. For simplicity, we assume that the the symbols can be one of 'a','b','c','d','e','f','g'. We also assume that the symbols frequencies are known.
Instructions

(a) Please copy the sample code (encode.c) from the locker. ('/mit/6.087/Lab2/encode.c')

(b) Please go through the code to understand the overall structure and how it implements the algorithm. In particular pay attention to the use of a priority queue and how the code tree is built from bottom-up.

Things to do:

(a) During each iteration, we need to keep track of the symbol (or composite) with the lowest frequency and second lowest frequency of occurrence. This can be done easily using a priority queue (a linked list where elements are always inserted in the correct position). The file 'encode.c' contains template code (pq_insert())that implements the priority queue. You are required to fill in the missing sections. Make sure you take care of the following conditions:(i) queue is empty (ii) new element goes before the beginning and (iii) new element goes at the end or in the middle of the queue.

(b) Symbols are removed from the priority queue using 'pq_pop()' function. In a priority queue, elements are always removed from the beginning. The file 'encode.c' contains template code to implement this. Please fill in the missing parts. Make sure you update the pointers for the element to be removed.

(c) Once the code tree is built in memory, we need to generate the code strings for each symbol. Fill in the missing code in 'generate_code()'.

(d) Finally fill in the missing code to free all resources and memory before exiting the code.

Output The program outputs two files 'encoded.txt' containing the encoded output and also 'code.txt' that displays the huffman code. Your output should match the reference values shown below:

symbol	code
a	10000
b	10001
c	1001
d	101
e	110
f	111
g	0

Table 2: Reference huffman code

Part C: Compressing a large file (Project)

Thus far, we have assumed the symbols and their frequencies are given. In this part, you will be generating the symbol frequencies from a text file.
Instructions

(a) Please copy the text file (book.txt) from the locker. ('/mit/6.087/Lab2/book.txt')

Things to do:

(a) Update encode.c to read from this file to generate the frequency of occurence.

(b) Generate an updated 'code.txt' and 'encoded.txt'

(c) Update decode.c (if required).

(d) Measure the compression ratio. Assume each character('1'/'0') in encoded stream (encodec.txt) takes one bit. Assume each character in book.txt takes 8 bits.

Output The decoded file decoded.txt and book.txt has to be identical.