ECE-368 Project-2 Report

- Abhimanyu Agarwal

Overview

The aim of the project was to work on Huffman compression and decompression. It expected us to generate a Huffman encoding scheme based on character frequencies in a file. On execution of huff.c, a stream of encodings should be generated which are written to a .huff file. Decompression reads the header and reconstructs the Huffman-tree taking the header information which is essential for decompression to work correctly. If the original file matches the .unhuff file generated , then we can say that decompression works correctly.

Compression - Approach

The file containing the original text was analyzed. The size of the file was computed. In addition, a frequency array based on the character occurrence was generated for every character in the file. After counting the weight, the characters are arranged in ascending order. The characters not included in the string are discarded and two characters with same weight are not swapped. Characters with same weight are sorted based on the ASCII value.

Now, the sorted list is put onto a linked list. Every linked list node would point to a tree node that would have a character and the weight. At the end of the list, a pseudo-EOF character will be added with the value of 1.We can just take first two nodes, merge them which implies adding their weights and putting that onto a new node which is also made to point to a tree node. Every-time two nodes with the smallest values are removed and are converted into one tree using tree merging function. This combined tree consists of the sum of the two lowest frequency values. The new tree generated is added onto a new node which is inserted onto the list.
Before, adding it onto the list, we have checks from frequency values which makes sure that the list remains in ascending order with respect to the weights.
This work is done until there is one node on the list, which would be the final tree.

We generate the Huffman Table which would be used for compression. In order to achieve this, we will use a 2D-array.We store the index number and the characters in this table.

The tree is traversed using Pre-Order Traversal. When a leaf node is seen, 1 is printed. When a non – leaf node is seen, 0 is printed. At the end of the sequence we append the pseudo-EOF character. The huff-table consists of the header information, that is required for decompression. In order to achieve, the final compression, bit wise operations are used. To reduce the number of bits while compressing and writing to the output file, we make use of masking.

Challenges Encountered

In my opinion, the biggest challenge was getting the accurate pseudo-EOF at the accurate place. In addition, I do feel that padding zeroes also did contribute to the cause in increasing confusion while checking the output that was generated due to compression.

Conclusion: I believe, my compression works as desired. It writes a couple of extra bits due to padding zeroes and leaves a space after writing the 8-bit encoding.

Decompression - Approach

Decompression works in the opposite direction compared to Compression. It is a complete reverse of compression. The algorithm looks for the pseudo-EOF character, once that is reached with in the compressed file while traversal, it would be break out of the decompression. At that point, we can say that the file is completely decompressed.

Challenges Encountered

One of the challenges was the tree-traversal because usage of feof() within a loop could lead to unpredictable behaviors that could potentially affect the functioning and execution of the loop.

Summary:

a) When decompressing we read bits and traverse the Huffman tree based on the bit value obtained on reading. Traversing is done to reach the leaf-nodes of the tree and read them.
b) Eventually we find a leaf-node representing some character.
c)When the pseudo-EOF leaf is found, the program / loop can terminate because all decompression is done by then.

Conclusion: Decompression fails, I get an output of my characters occurring in a random order. However, my tree is formed which comprises of all characters in the correct traversal order which implies my decompression partially works. I believe, cause of the failure is incorrect interpretation by my code while traversing the tree.

Related outputs obtained on huff.c() execution:

Input Sample Filename (.txt)	Input file nature	Time taken (s) (User time)
Test0.txt	Small (=1kb)	0.000 s
Test1.txt	Small (=1kb)	0.001 s
Test2.txt	Small (=1kb)	0.001 s
Test3.txt	Large(> 1kb)	0.050 s
Test4.txt	Large(> 1kb)	0.094 s
Test5.txt	Large(> 1kb)	0.147 s