ECE 368 Project 2 Report

Ben jiang

This project requires an implementation of Huffman encoding to compress and decompress files based on the commands. The compression algorithm scans the input file and counts the frequency of each character that is used in the ASCII table. Then it generates a Huffman tree based on the frequency that each character was used in the file. A Huffman code table is generated using the Huffman tree for each character used. And the frequency array, the number of characters in the file, along with the Huffman encoded translation of the input file is written to an output file. The decompression algorithm first reads in the header and assigns the frequency array and the character count to the correct variables. Then the Huffman tree is rebuilt using the same methods in the compression algorithm. The data portion of the compressed input file is read in one bit at a time to traverse through the Huffman tree and prints out the character at the leaf nodes when one is reached until the end of file is reached.

Compression:
This portion of the project is all contained in the huff.c file. This part contains the most important functions of the entire project, some of which will be reused in the decompression portion of the project. First, the entire input file is scanned character by character to generate the size 256 frequency array that contains the number of times each character is used in the file with each element representing each character with its ASCII value as its index in the array. The frequency array is then passed to a list generation function that creates a linked list of specially defined structure that contains a tree node each representing a character that is used in the input file. The function is written in a way that each new node added is checked to be placed in the proper location within the linked list so that the linked list is always sorted in an ascending manner with regards to the frequency. The linked list is then passed to a tree generation function that takes the two nodes with the smallest frequency merging with a newly generated parent node that has a frequency value of the two daughter nodes combined. The new node would be added into the linked list while maintaining the order with frequency. This process is repeated until there is only one linked list node remaining, which would naturally contain the head node of the Huffman tree as its tree node parameter. The Huffman tree is then passed into a Huffman code generating function that goes through the entire tree recursively while saving its traversal steps with “0” representing a movement into a left sub tree and a “1” representing a movement into a right sub tree. These steps along with the character parameter of each leaf node that is reached are saved in a 2D integer array with the number of unique characters in the file as the row number and the height of the Huffman tree as the column number of the array.

When writing the output file, a header is first written with the frequency table first, followed by the total number of characters in the prison. This is not the most space efficient, but it comes with the benefit of the convenience of implementation both in writing and reading since it would always have a fixed length, and the same functions

in the huff.c file could easily be reused in the unhuff.c file for tree reconstruction. After all, for larger test cases, this header doesn't have a big impact on the size of the compressed file. The input file is then read one character at a time and encoded by checking the Huffman code table for a binary representation of each character till the end of the file. Finally, padding is added at the end of the output file to ensue 8-bit alignment.

Decompression:

This portion of the project is relatively simple compared with the compression portion. First, the header is read in and the frequency array and character count are written to the proper variables. Then the list generation and tree generation functions can be taken from the compression portion of the project to rebuild the Huffman tree for decoding the compressed file. Then by reading the compressed file one bit at a time and traverse through the tree continuously, with a "0" representing stepping into the left sub array and a "1" representing stepping into a right subarray. This is easily achieved with a recursive function actively checking the parameters of a node that it reaches in the process. When a leaf node is reached, the character parameter of the node is then written into the output file and the traversal is reset to the head of the tree and the total character count is decremented by one. This is done till the character count reaches zero, which means the entire compressed file is decoded into its original form.

Table 1: original vs compressed file size comparison

	Original File Size (bytes)	Compressed File Size (bytes)	Compression Ratio
Text0.txt	4	1000	250
Text1.txt	8	1000	125
Text2.txt	155	1080	6.97
Text3.txt	3077k	2259k	0.73
Text4.txt	6153k	4517k	0.73
Text5.txt	9229k	6775k	0.73

All test cases created matching decompressed files and original input files. As shown in the table above, the smaller test cases have larger compressed files than original files. This is due to the header taking up a lot of space in the compressed files.

One notable issue I experienced when completing this project was an issue with regards to the string modification functions to generate the output file name. The bad output file names generated caused the output file pointer to not open properly and the encoding function was unable to write data into the output function. This was very tricky to debug, and it took me hours before I was able to recognize it. Besides this, the program was overall very straight forward.