NO SPECIAL COMPILATION FLAGS NEEDED

Jonathan Morlen

ECE 368

10/22/19

Huffman Coding Report

<u>What I Did:</u>

Huff.c

1. Iterated through input file, counting occurrences of characters and storing the frequency data in an array.
2. Trimmed the array down to include only characters that are present in the input file.
3. Sorted frequency array in ascending order, using Bubblesort
4. Transformed array into a doubly linked list of special Node struct's. These structs function as elements in a doubly linked list as well as nodes in a binary tree for easy transitions between the two data structures.
5. Grabbed the first two (lowest frequency) nodes from linked list, paired them together into a parent node, and inserted parent node back into the linked list at an appropriate position to maintain sorted state. Repeated until the doubly linked list only contained one node, the root of the Huffman tree.
6. Used the Huffman tree to generate bit encodings, storing these encodings along with the characters they are attached to, in a separate doubly linked list.
7. Used the list of bit encodings, the name of the input file, and the Huffman tree to once more iterate through the file, find the bit encoding for that character, and write it to the output file.

Unhuff.c

1. Iterated through input file, reading the header and reconstructing the Huffman tree.
2. Iterated through the remaining portion of the input file using the Huffman tree to decode the bit sequences into characters, writing those characters to the output file.

<u>Challenges:</u>

This was by far the most time-consuming programming project I have encountered in my academic career, so I will highlight a few of the roadblocks I hit on my way to completing this assignment.

First and foremost, the biggest obstacle I faced was determining exactly what the Pseudo-EOF signal/ ”character” actually was. For the longest time, I was stuck thinking how to represent it in a single char given the fact that it was entirely possible for every char to be used in the input file. (I sincerely hope the project is graded with the full 255 characters, as I put a lot of effort into getting it right). I finally managed to figure it out and represent all characters as 9-bit sequences instead of 8, with the first bit acting as an indicator for the Pseudo-EOF. I stored my header in the encoded file as the full Huffman tree in pre-order traversal with a 0 indicating an internal node and a 1 indicating a leaf node. After a 1 was detected my program would then read the next bit and determine whether or not the Pseudo-EOF had been triggered or not based on its return value.

Second largest roadblock was more general and widespread, being my own failure to realize that ASCII will include characters from 128 – 255. I had originally written the program(s) only accounting for 127 possible characters, using char’s to store all characters. Once I realized my mistake, I had to make quite a few passes changing char’s to int’s. This had a pleasant side-effect of changing my malloc’s from depending on the size of the variable type, to depending on the size of the dereferenced variable name, reducing complications when frequently switching types. Along the same vein, I moved from calculating the file size as an int, to calculating it as a long to account for larger files, to not calculating it at all and just fgetc’ing my way through the input. This was beneficial because I no longer read the entire file into a char*/int* in one massive chunk, I instead went byte by byte, which is much more efficient and saves memory from less malloc’ing.

A smaller, yet present problem I encountered was how to read and write bits, in 8-bit chunks. I wrote a pair of bit/byte writing and reading functions for huff.c and unhuff.c respectively. These functions made use of a char*/int* buffer that was declared at the highest level, sharing its information with every read and write operation within the program. These functions made use of an variable level, which indicated how “full” of bits the buffer was. Once this indicator variable reached a value of eight, the byte was written to the output. Or in the case of unhuff.c, once the level reached zero, another byte was read into the buffer. Each xxxxByte() function called the xxxxBit() function 8 times, maintaining the level and byte variables to easily transport this data. Figuring out bit shifting and masking was also difficult given that we have not learned that at Purdue yet, but it was a good challenge to learn it for this project.