David Pimley
ECE 368
10/24/17

Project 2: Huffman Coding and Decoding – Final Report

The overall program can be broken into two parts. The first part of which is the compression of the input ASCII file, with the second part being the decompression of a binary file into the same ASCII file that was compressed.

The first part, *huff.c*, takes in a normal ASCII file and compresses it to smaller file that contains the same data, i.e. lossless. The way I wrote *huff.c* I tried to make it run as fast as possible and as clear to read as possible. To do this I broke up the program into multiple functions. The program first starts by reading in the input file and storing the frequency counts of each character that are present in the input file until the EOF is reached within the input file. These frequency counts are then passed into a function, *create_heap*, that creates an initial priority queue / heap where each index in the array is a pointer to a t_node. A t_node in this project is just a tree node that can hold a character and a frequency. To help make my code more readable I created a function, *create_t_node(),* that would, when given a count and a label (character), would create a t_node and return a pointer to it. Each time a t_node is created it would then be inserted, using *insert_heap(),* into the priority queue / heap into its correct location. This function would work by first setting the end of the array (bottom of heap tree) and sifting the value up while its parent is greater than the value of the node being inserted. The important thing to note is that for this priority queue / heap, the min is at the top so therefore it would be defined as a min heap. Once the priority queue was fully created with each node pointer in its correct position the Huffman tree was then created. To do this three different function were required. One function to actually facilitate the creation of a Huffman tree (*build_huff_tree)*, a function to remove the minimum from the priority queue (*remove_min)*, and one function to sift down the head of the priority queue into its correct position (*sift_down)*. The main Huffman function would first extract the two smallest nodes (based on frequency count) from the priority queue and create one last node to connect these two nodes. The left and right pointers were then set to the two min nodes. To ensure that after removal the priority queue would still be correct the sift down function was utilized to repair the heap. This function would work by setting the last element in the array at the top of the heap and sifting the node down by checking each child of a root node and determining which one was smaller. The smaller node would then be switched with the parent to ensure the smallest node became the parent. The nice thing about using heaps for this was that the minimum value is ensured to be at the beginning of the priority queue which not only makes removal of nodes faster, but easier to understand since the 0 index of the array is removed each time a minimum value is needed. The Huffman tree creation program would then insert the top node (parent of the two minimum nodes that also holds the combined frequency of the min nodes) into the priority queue. This process would then repeat until there is only 1 node pointer left in the priority queue in which case this pointer would then be returned. One last function I needed was a function, *create_huff_table(),* to print all of the Huffman codes for the present ASCII characters into an array of structures. The structures would not only hold the length of each Huffman code but the code itself stored in an integer. What is meant by this is that instead of having a code look like 10100 and storing this as an integer, the equivalent would be to store this value as an integer value of 20 with length 5. Once the table was created functions were needed to actually print to the output file. The first thing to be done was to print the header (*print_header)* of the file as a preorder traversal of the Huffman tree that was created earlier. Each "connector node" was printed as a 0. If the node was a leaf node that contained a character form the input file, this would be printed out as an ASCII 1 followed by the actual ASCII character. For the EOF character the function would only print out a 2 to signify that this is the EOF node. Once the header was printed the last function, *write_compressed_data(),* compressed the data from the input file to the output file. This last function would take in a character from the input file and find the corresponding ASCII character in the huff table that contained all of the codes created form the Huffman tree. The function for normal character, i.e. not EOF, would take in the code and shift the bits to the right so that the most significant bit of the code would be in the 2^0 position. The other values were then masked off and then this bit would be added to the buffer. Once the buffer became full, (8 pushes into the buffer), the buffer was then written out to the file. The only tricky part about this function was writing out the pseudo EOF. If the character read in from the input file was EOF, this let the function know that the pseudo EOF had to be written out to the compressed output file. At this point in the function if the number added to the buffer was 8 after the EOF code was added then the function would return. However, if the buffer still has empty positions left in it, the buffer would then pad on 0's to the end of the buffer until it was full and then would print.

The program created from *unhuff.c* was much simpler than that of *huff.c* because the tree did not have to be created from frequency counts again. Instead the tree was created form the header written out from *huff.c*. Therefore only two new functions needed to be written in *unhuff.c*. The first of which created the Huffman tree from the header information at the beginning of the compressed file. This function, *read_header(),* would work by creating a "connector node" each time a 0 was read in and creating in a leaf node each time a 1 or 2 was read. The meat of this program was to create a function that would read bits in and assign ASCII characters based on those bits. This function, *print_file(),* would run while the EOF flag was not set and would traverse the Huffman tree until a leaf was reached and would then print that character to the output file. The bits for each byte were shifted off in a very similar manner to that of the first compression program. Bits were first read into a buffer and shifted right to the 2^0 position where the rest of the bits would then be masked off. The remaining bit would then be considered the direction to go in the tree where a 1 is right and a 0 is left.

Overall, the programs have an emphasis on the overall speed rather than the compression ratio. Below is a table of the results from running huff and unhuff.

FILE	TIME		COMPRESSION INFORMATION		
	huff (s)	unhuff (s)	Compression Ratio	Original File (bytes)	Compressed File (bytes)
text0.txt	0.001	0.001	0.444	4	9
	0.000	0.000			
	0.001	0.001			
text1.txt	0.001	0.001	0.276	8	29
	0.000	0.000			
	0.001	0.001			
text2.txt	0.001	0.001	0.923	155	168
	0.000	0.000			
	0.001	0.000			
text3.txt	0.115	0.216	1.348	3.1M	2.3M
	0.111	0.213			
	0.004	0.002			
text4.txt	0.234	0.415	1.356	6.1M	4.5M
	0.226	0.404			
	0.009	0.009			
text5.txt	0.343	0.629	1.358	9.1M	6.7M
	0.328	0.613			
	0.013	0.009			
vlrg.txt	20.410	36.887	1.359	541M	398M
	19.523	36.199			
	0.621	0.645			