ECE 368 Project 2 Report

Sthitapragyan Parida

This project demanded a huffman encoding scheme based on character frequencies in a file. In short, huff creates a huffman tree and encodes the file using the tree. Then it writes a .huff file with the header info containing the tree and and the encoded file text. Unhuff does the exact opposite by reading the header, reconstructing the hufftree from the header and decoding the .huff file to generate a .unhuff file which matches the original text.

Encoding

The basic approach used in huff.c is the following: First, the file was analysed to generate the frequencies of each character in the file. This was stored in a character array of size 128 to store the frequencies. Then this array was passed to a createHuffTreeCode. This function generated a HuffNode for each alphabet and assigned each of them their respective frequency. Afterwards an extra Node with value EOF and frequency one was added to the list to assist in terminating decoding operation in unhuff.c.

Each HuffNode contains four elements: character, frequency of character, left child and right child. Further a container struct was created to house each HuffNode so that a priority based linked list could be created out of the HuffNodes with the node having the least frequency being the head. This was done to assist in the combine nodes step pf the Huffman algorithm. After creating a frequency based linked list for the HuffNodes, the list was processed in the following steps:

1. The head and the link right after were combined to create a new HuffNode with frequency equal to the sum of the individual HuffNodes.
2. This new node was inserted into the priority linked list.
3. The process was repeated till there was one element in the linked list. The Huffman Tree
4. While combining leaf nodes to create internal nodes the character with ASCII value 0 was used to dummy fill the character variable in the internal HuffNode.

After obtaining the Huffman tree, it was processed to obtain the encoded bit strings for each character in the file. To do so a Huffman table was created with l rows and h columns, where l is the number of unique characters in the file and h is height of the Huffman tree. This table was filled using recursive techniques.

Before encoding the file, the Huffman tree was stored in the header of the output file suing the technique of in order transversal. Thereafter, the input file was scanned character by character and each character was encoded into its Huffman bit representation. To facilitate with writing one byte at a time to the file, a buffer of max size 50 was implemented using a char array. Each bit of the output file was stored in the buffer and everytime the the buffer size exceeded 8(a byte was available), a byte was written till the buffer size reduced to a number lower than 8. After encoding the file, an extra encoded EOF was written to the file to signal termination of decoding loop in unhuff. Finally the buffer was checked for remaining bits and the remaining bits were padded with zeros to bring the total to a byte (8 bits) and it was written to the file too. This concluded the Huffman encoding part of the project.

Decoding

The Huffman Decoding part of the project had a simpler implementation. Initially the Huffman tree was read from the header of the encoded file using recursive transversal. Then the file was read byte by byte. Each byte was processed bit by bit. For a 0, the current node changed to its left child and for a 1 to its right child. If a leaf node was encountered, the current node was reset to the root node of the tree while the character value of the Huffman tree leaf node was output to the .unhuff file. The loop continued till the decoded character matched EOF, wherein the file was closed and the decoding process was complete.

Note: As the header uses characters to output the ones and zeros the compression ratios for small files is higher than 100 %. However, the algorithm achieves an average of X% compression for bigger files.

Original File Size	Compressed File Size	Compression Ratio	Original Matches Unhuff?
13b	21b	1.615	Yes
3.6kb	2.1kb	0.5833	Yes
531.8kb	299.1kb	0.56	Yes

As is visible, the compression ratio is low for big files. However, due to the additional header, the compression fails for lower sized files. This can be fixed by using bit representation of the ones and zeros in the header than characters.

Running & Compiling The Program

```
gcc -o huff huff.c
gcc -o unhuff unhuff.c
./huff <filename>
.unhuff <filename>.huff
diff <filename> <filename>.huff.unhuff
```