Matthew Wen

The Huffman coding algorithm is a greedy algorithm that requires a binary tree to compress and decompress a file. As a result, I designed my algorithm to become as efficient as possible, while also taking the account the compression. Because compression in general requires the user to recall the original information with less memory, it is important to carefully structure out how the compression file (huff) will give to the uncompress algorithm (unhuff). As a result, the huff executable must be able to create a header with the least amount of bytes in order for the unhuff executable to retrieve that information. In this report, I am going to discuss how my binary tree compresses the data compared to a raw ascii file, then I am going to discuss how I created my header. Afterwards, I am going to discuss how the huff executable evaluate the data from the original raw file, then how unhuff decompresses the file to its original form.

The layout of the binary tree is important because the direction in terms of 0 or 1 is being stored in the compressed version of the file. Normally, characters in ascii are stored to have 8 bits; therefore, on average, each character takes up 8 bits for large inputs because the amount of bits occupied is constant. If the purpose of huffman is to compress the data, then as the number of inputs increase, the average amount of bytes taken up by each of the characters should be less than 8. The average can be calculated by multiplying the frequency of each character by its depth of the tree divided by the number of characters in the file. The depth of the binary tree is set up based off how the input is organized. For example, if all the elements are sorted, then the tree depth would be 256 at most, or the number of unique characters used in the file. The issue of having a depth greater than 8, especially 256, is that a character would use more space than required. To prevent this, I create an array with nodes with values. I sort all the elements by insertion; the time complexity is not important because the size will be no greater than 256. I then replace two of the smallest values with a node that contains the summation of its weight, weight being defined as the frequency of character in file. I then perform insertion again with the node, then repeat the process where I take two of the smallest values or nodes, then recreate another node. The time complexity would be defined as $O(n^2)$ for the initial insertion sort, then $n^2 + (n-1)^2 + (n-2)^2$ which would increase the complexity to $O(n^3)$. However, since the number of elements is decreasing the time difference would be insignificant. Because insertion sort is performed, the memory complexity to sort the weight would 1, but the number of nodes would be allocated, which therefore increases the memory complexity of $\log(n)$, where n represent 256 as the maximum. This method insures the average depth of the tree would be less than 8. The worst case scenario is if all the weights or occurrence of all 256 characters are the same, therefore the depth to reach all the nodes will be 8, making in no more efficient than using a raw file. The average case is if the characters with the most occurance stores less bits inside the compressed file, and characters to minimal to no occurrence stores more than 8, since it is a valid trade off if the character is not used often.

The binary tree used in huffman must be recreated, and the header of a huffman file is used to create the tree. My initial thought is to restore the frequency and the character inside the header. This uses too much space because the header size would be dependent on (sizeof(int) + sizeof(char)) * number of characters used. To avoid this issue, I used bit encoding to store the location and

direction of a character in a tree, since the frequency of each character occurring is not important. This is done by encoding 0 until it hit a node that stores a character; the hit is indicated by a 1. After hitting the value 1, an 8 bit binary interpretation of the character follows afterwards. Each 0 represents a new node that is left from the head. After setting value at the node, it continues the process with 0, where it allocates a new node represented by a 0 until it hits a 1, where it encodes it's next character. The process continues until the tree is fully interpreted pre ordered. This minimized my text3 header size form 500 bytes to only 106 bytes.

For the huffman executable to determine what bytes to encode inside the compressed file, it stores the location of each of the leaves inside an array with 256 elements. Therefore, instead of scanning through 256 elements to find the location of the element, or to prevent the use of using additional memory trying to find it recursively through an unsorted binary tree, the character in decimal form is its location inside the array; the location is determined by being recursively set after the tree is done being created. The program contains a uint8_t integer that stores the bytes of the location of each. If all 8 bits of that data type is used, then it writes it inside the file. As a result, padding is expected at the end of the file.

For the huffman executable to decompress the file, it recreates the binary tree expressed above; it looks at a sequence of 0s and 1. To rephrase, 0 means to allocate a node on the left of the head, and 1 indicates to allocate a node that stores the character value; the character binary sequence follows after the 1. For each character, the time complexity will be 8 * number of characters since 8 is the maximum average for depth of tree. Therefore, the big O notation will be O(N). The program also stores a stack where it reads 8 bits when it is done reading the current 8 bits in the stack. The decompression executable have to iterate through the binary tree because the compress file only reference each character based off its location. For my decompression software, every time it reaches a node with a value, it returns back to the root to reevaluate the location of the next character and so on.

| text0.txt | | | text1.txt | | | text2.txt | | |
|---|---|---|---|---|---|---|---|---|
| | Runtime (sec) | | | Runtime (sec) | | | Runtime (sec) | |
| Ratio | huff | unhuff | Ratio | huff | unhuff | Ratio | huff | unhuff |
| 7 | 0 | 0 | 4.625 | 0 | 0 | 0.94 | 0 | 0 |

| text3.txt | | | text4.txt | | | text5.txt | | |
|---|---|---|---|---|---|---|---|---|
| | Runtime (sec) | | | Runtime (sec) | | | Runtime (sec) | |
| Ratio | huff | unhuff | Ratio | huff | unhuff | Ratio | huff | unhuff |
| 0.742 | 0.105 | 1.622 | 0.7348 | 0.227 | 3.133 | 0.736 | 0.315 | 4.620 |