ECE 368 Project2 Report

Yunlei Yan

0028847390

1. Approach

   To compress the file, I first read through the file to generate the frequency of each character. Using the frequency, I build a Huffman tree. After that, I stored the code of each path to leaves into a array for further use. Read through the file again and find the corresponding path of each element and store it in the buffer. Whenever the buffer reaches 8 elements, we put them into the binary file.

   For decompression, I stored the paths and their corresponding character into the binary as well. During depression, I first retrieve the array where I stored my "decoder" and then read the rest of my binary file and decode the numbers.

2. Efficiency analysis

   a) Time to finish the algorithm

	Text1	Text2	Text3 150000lines	Text4 300000lines
Compress	<0.000001s	<0.000001s	3.26s	7.38s
Decompress	<0.000001s	<0.000001s	109.21s	212.33s

   For compressing the file, building the tree will only take O(n) time. However, when I was trying to encode the file and load it into the output, it will generally take O(num of elements * num of leaves). Since the number of elements is much greater than number of leaves, my compress algorithm is also O(n).

   For depression, it is still O(n) as both algorithm go through the whole file to finish their job. In this case, the running time is relatively fast comparing last time trying to sort the whole file.

   b) Space takes by each algorithm

   For both algorithm, the space it required will depend on how many kinds of characters in the file. So, the maximum space will 256 which is the number of ASCII codes.

3. Compression performance

	Text1	Text2	Text3	Text4
Ratio of compression	8.875	2.277	0.734	0.734

From the table we can easily identify that for rather smaller files, the program is ruin everything by increasing file size after compression. This is because I am printing the tree into the file as well. Small files with distinct characters could possibly create a tree larger than 25% of the size of the file which, as a result, make the ration bigger than 0;

However, as the size of files grow, the ration tends to approach a constant which is around 0.734. This indicates that my program can save more than 25% of the space taken by the original file.

4. Makefile

In my makefile, odd test cases are used to perform huff operations and even test cases are for unhuff operation.