ECE 368

Project #2 Report

By- Kshitij Jain

For this project we had to create a two programs- huff.c and unhuff.c , which compressed the file and uncompressed the file respectively.

I obviously started with the huffing part of the project since the unhuff part is basically a follow up to the first part.

So, for my huff.c, I took a lot of help from the input files given to us in ECE264, when we had to basically, do just the unhuff part of the project. I greatly used these files to test the output files generated by my huff.c. These file were all of small sizes and once my huff.c was working on these files, I tested it on the input files given to us and was able to generate the correct output files, which I later tested by unhuffing those files.

In huff.c I went the following way-

1. Read input file and stored the characters read and their frequencies in two arrays.
2. Used these two arrays to create a linked list of input characters.
3. Sorted this linked list based on frequency of the characters.
4. Created a huffman tree from this linked list, using the insert and delete helper functions.
5. Traversed the tree created, post-order and created the header part(used for decompression) of the output file.
6. Wrote the actual data of the input file in binary(this wasn't really binary, but binary data extracted and then converted to chars).

For me, unhuff.c was pretty easy to do, as I had done it previously in ECE264. I basically reused a lot of code from 264. Here's my workflow for unhuff.c-

1. First, I read the header part of the input file generated by huff.c and using a stack, I regenerated the huffman tree that was used to compress the data.
2. At the end of the header file I had also given info about the number of characters in the input file. I read this and stored it.
3. Now, with the help of a function called readbit, I read the actual data in binary from the input file, converted it to actual characters using the huffman tree and wrote it to an output file.

In the whole project, I believe the trickiest part for me was to write the actual data in binary. Most people did it using bit masking, but I couldn't really understand how I was supposed to do it so I just did it using a different method, which has been described in detail in my huff.h.

The time and space complexity of my program are as follows-

Time complexity-

* O(n)+O(n)+O(nlogn) = O(nlogn)(reading input file + sorting chars based on freq + greedy algorithm to create huffman tree)
* O(n)+O(nlogn) = O(nlogn)(reading input file + creating huffman tree)

Space complexity-

* O(k) + constant for huff.c (k is number of different characters in input file)
* O(k) + constant for unhuff.c(k is number of different characters in input file)

Following are the compression ratios and the time taken by my huff.c and unhuff.c for the input files, provided to us by-



For compression ratios less than 1 the compressed file size was actually greater than the input file size. Therefore, we can deduce that huffman coding is good only for large files. The taime taken by both huff and unhuff grow fast with input size, while the compression ratio grows really slow with respect to input size.

Flags used and testing info for compilation and running the program:-

* Please use the -lm flag for compiling huff.c
* huff.c takes one input, which is the input file name
* unhuff.c also takes one input, which is the input file name