Matthew Bolda (PUID 0028979907)

ECE 368

10/24/19

Project 2 Report

In this project, we had to use Huffman coding to compress and then later decompress a file. I will briefly explain the process I used for both compression and decompression. Also, I will highlight the obstacles I encountered and how I was able to get around them.

Compression:

Let's begin with compression, Huffman compression is a method that uses unique bit codes to represent characters, with the most frequent characters having smaller bit codes to make them use less space. To implement this, I first needed to find the frequency of each character in the file. I created an array large enough to fit every ASCII value, using this array I went through the file and added a count for every ASCII element. Once the frequency array is filled out, I created a Linked List, every node had an ASCII value, frequency, and a link to the next node. In this list I created an EOF node that had a frequency of one, this would come to help me later. After the Linked List was completed, I created a Heap using the heapify function we learned in class. This heapify function would continue to be useful as I would remove the two least frequent Heap Nodes and build a new node with links to both of those nodes. Slowly the Heap will turn into just one node and that node will be the Huffman Encoding Tree. This was the hardest part for me and I had several problems getting the heapify function to work.

From the Huffman Encoding Tree, I created a table of characters in the file and their corresponding bit codes. This table would be used to finally compress the file. The first thing I did was write a header into the file, this header was the frequency of every character in the original file. I then collected the character from the original file, found the character in the table, and printed the bit code bit by bit for each character in the file. I originally made the mistake of printing out the bit codes as characters so my "compressed" file was always larger than the original. Making sure to change the output file to the original name plus ".huff" I was able to finally compress the input file.

Decompression:

That brings us to decompressing, the most important part was to read the header of the compressed file. After reading this header you could construct the same exact Huffman Encoding Tree as used to compress the file. With the Huffman, Encoding Tree recreated you could read bit by bit the compressed file. If the bit was a ‘1’ you moved to the right node if it was a ‘0’ you moved to the left node. Once you reached a leaf node you printed the ASCII value of that leaf node into the file and returned to the top of the tree. You would stop reading after you got to the leaf node that is your pseudo-EOF. Originally reading the bits one at a time was difficult, I eventually learned to use a mask so that I could read just 1 at a time easily. Another obstacle I had was determining a good pseudo-EOF, for a while I picked ASCII value 0, but I have been experimenting with having leaf nodes that have both ASCII values and a Boolean that says if it is EOF or not. This means that every leaf node will have an ASCII value and a Boolean that says it is not the pseudo-EOF, except for the pseudo-EOF that will have an empty ASCII value but a Boolean that says it is the pseudo-EOF. That way when I am reading bits from the compressed file and reach a leaf node, if the leaf node is not the pseudo-EOF get the ASCII value and print it to the file. The filename for the uncompressed file was changed to be the name of the compressed file plus “.unhuff”. To test if it was properly decompressed I used the bash diff command comparing the original file to the unhuff’d file.

Compiler:

For some reason when I used the huff.o and unhuff.o files given to us I received segmentation faults for the larger files. However, when I use .o files I've created using the gcc compiler line in the project pdf I do not receive these segmentation faults. The compiler line I used to create my .o files was the same as the project manual pdf:

```
gcc -Werror -Wall -O3 huff.c -o huff
gcc -Werror -Wall -O3 unhuff.c -o unhuff
```