PROJECT 2 REPORT

Name: Chul Woo Park
Homework ID: 4600081

1. Description of Algorithm

I implemented compression (huff) and decompression (unhuff) program based on the Huffman Encoding algorithm. Firstly, my huff program almost exactly follows the Huffman Encoding algorithm, which makes a tree that each node stores the information of ASCII character and the frequency (how many times it appears in the input file). In order to create such tree, my huff program firstly reads the input file and creates an array size of 256 that contains the frequency. The name of the array is asciiCount. The index number of this array is the ASCII value of the character and the value stored in that index is the frequency. For example, if character 'g' appears 3 times in the input file, asciiCount[103] would be 3. In this way, my program knows which character appears how many times in the input file.

Then, I use this array to start building the tree. My array firstly creates a linked list based on this array. The important thing is that the node is inserted into the linked list and also being sorted. Basically, the corresponding node gets inserted into the right place. When all the information in asciiCount array is converted into a linked list, all the characters with non-zero frequency will be in the linked list. Creating the linked list is done by the addNode function. If the information in asciiCount array is sent to the addNode function, it creates a node with it and goes through the linked list until it finds the right place to insert. In this way, my program can keep the linked list always sorted while inserting the nodes. With this sorted linked list, my program builds the tree with it.

The algorithm of building the tree starts with picking the first two nodes in the linked list, which are the lowest frequency characters. Then, it builds the tree node that has these two nodes as child and the count is assigned with the sum of these two nodes' count. This tree node has a label of 1000 in order to indicate that it is a non-leaf node, that has two children. I just used 1000 because ASCII is only from 0 to 255 so that no leaf-nodes will have a label of 1000. Then, this tree node created with two selected nodes as children will be re-inserted into the linked list because the count has been updated so that it needs to be sorted again. After multiple recursions of buildTree function, my program will have the final tree that is essential to my compression method.

Since my program has the tree, it needs the table that has an information that which character has which path in the tree. This table is needed in order to write the compressed information on the output file. Therefore, I created a 2D array in my algorithm. The first index of this 2D array is a ASCII value of the character and the path is stored in the second index starting from 0. When entire path is recorded, it is indicated by integer value of 7 that it is end of the path (7 is randomly selected value). With the tree and table, my program starts to write on the output file. The output file of my huff program contains two information, which are header and contents. Firstly, header part includes the tree information in pre-order traversal. Since not all information is in 8-bit (1 byte), I used the buffer with writeBit function. When non-leaf node is found, it writeBit 0 and when leaf-node is found it writeBit 1 and also the ASCII value of that character, which is in 8-bit. The writeBit function in my program has a buffer named “curbyte” and it writes the curbyte into the output file when it is filled with 8 bits. As I just mentioned, the header information won't always be in bytes so that my program padZero at the end.

After my header information is succesfully written to the output file, it starts writing the contents of the input file. Basically, my program reads the input file character by character and writes corresponding path in the tree. In addition, my tree has an extra node that indicates the EOF. Important thing here is that I assumed that null character is not in the input file because I inserted EOF node in the tree with the label of 0. Therefore, after writing all the contents in the input file as a compressed form, EOF path is inserted at the end. In order not to lose some important bits that are remained in the buffer, my program padZero at the end.

With this huff program, I also implemented the corresponding unhuff program that decompresses the compressed file that my huff program created. My unhuff program is fairly simple. Firstly, it reads the header part of the compressed file. This is done by readHeader function. It uses recursive function calls in order to connect two leaf-nodes to the non-leaf node. I also use buffer in order to read the bits and the algorithm is really similar to writeBit function in my huff program. When finished reading the header part, readHeader function returns the tree. At this point, unhuff program has an access to the tree that my huff program created so that it is

ready to traverse the tree.

Finally, my unhuff program reads the contents part of the compressed file and readContents function does this. Basically, it traverses the tree until the EOF node I intentionally put in the huff program is detected. When reading the bits, it goes to right of the tree if bit found is 1 and left if bit found is 0. It prints the label of the node when it hits the leaf-node which contains the character. When all these methods are done, my unhuff program successfully decompresses the compressed file. Therefore, output from my unhuff program is exactly same as the original file that was compressed by my huff program.

2. Size Comparison Between Original File and Compressed File

File Size	basic	gophers	para	prideandprejudice	woods
Original File	5 bytes	14 bytes	819 bytes	704175 bytes	133 bytes
.huff File	9 bytes	19 bytes	481 bytes	396800 bytes	106 bytes
Compression Ratio	0.5555556	0.736842	1.7027	1.77463458	1.254717

It seems that the compression ratio is higher for bigger size of input files. The reason is that even though the size is compressed in the contents, my huff program still needs to write the header information that contains the tree. Therefore, it makes sense that header has an even bigger size than the contents for small size input files. I would say that there will be no meaning of compressing the small size of files based on my huff program. However, for bigger size files, it should compress quite decently.

3. Time

Since both of my huff program and unhuff program compresses and decompresses all of the 5 files above immediately. I think time won't be a problem for even bigger size of input files.

4. Compile Commands in Makefile

My Makefile has several recipes that might help in compiling and even checking the outputs.

For huff:
make all : compiles the huff program and creates an executable named “huff”
make test : execute “huff” with several input files (should be modified to test other input files)
./huff [input file]: execute “huff” with specific input file
make memory : check valgrind errors for the huff program (should be modified for specific input files)

For unhuff:
make all2 : compiles the unhuff program and creates an executable named “unhuff”
make test2 : execute “unhuff” with several input files (should be modified to test other input files)
.unhuff [input file]: execute “unhuff” with specific input file
make memory2 : check valgrind errors for the unhuff program (should be modified for specific input files)

In General:
make compare : compares the original files and decompressed files done by my unhuff program
make clean : remove all executables, core dump files, and output files

* compilation flags are indicated on top of my Makefile