Huffman Code Generator Thingy

Designed and implemented by Matt Brown and John Elson

Final Project for Coding and Information Theory

**Purpose:**

This program is intended to create a Huffman code for characters. That is, the average length of the code will be at a minimum and the code itself will be both instantaneous and uniquely decodable. In this case, a training input is taken either from the keyboard or from a file and an alphabet is constructed using the characters from the input. Only characters from the training input are recognized and encoded/decoded. The system creates a Huffman tree using the frequencies of the characters from the input. The characters that have a higher frequency, that show up the most, have a higher probability of being encoded. The Huffman tree is traversed and a code is assigned to each character based on the number of times the character showed up in the training input.

Now, input can be encoded using the Huffman codes and decoded using the same codes. Instructions for the program are included below.

**Instructions:**

We have included two versions of the program. We had hoped to have a graphical user interface finished by this time, but we fell short of our intended goal. The GUI isn’t functional, but you can run the code from the appropriate folder to see what we completed so far. However, a text-based version of the program runs perfectly with our test cases. The text based version is the one you should run.

To run the program, navigate to the Huffman folder and compile the java files. This project was coded with Java 7 update 45. The HuffamnGUI requires JavaFX, which should be included in the version of Java mentioned earlier. All of the files are part of a custom package called Huffman. If there are errors during compiling, you can comment out “package Huffman;” in every file. We had some issues while compiling with a console, but it compiled fine with the Netbeans and Eclipse IDEs. Make sure the files are in a directory called “Huffman” before compiling. To run the program, run main.class.

Upon running the program, you will be presented with some options. 0 exits the program.

1. Read data
2. View Frequencies
3. Encode Message
4. Encode from file
5. Decode from file
6. Read from file

Option 1: This reads training data from the keyboard. The user types in some characters and upon pressing enter, the data is used to generate a Huffman tree and the tree is traversed to generate the codes. \*\*IMPORTANT\*\* The characters used to train the program are the only characters that can be encoded. If a ‘Z’ never shows up in training input, the program will not encode a ‘Z’. This goes for both uppercase and lowercase versions of the letters and for symbols.

Option 2: This will print out each character, its frequency and its code.

Option 3: This takes input from the keyboard and encodes it using the training data supplied earlier. If there is a character that is not found from the training data, those characters are output to the screen. The resulting code from the input is printed to the screen. The code is then decoded and the result from the decoding process is output to the screen as well. This is just to show that the encoding and decoding process works correctly.

Option 4: Here, you type in the name of a file in the same directory as the program. For example, if there was a file called “test.txt” in the same directory as the program, you would type “test.txt”. The file is loaded and its contents are encoded using the codes generated from the training data. The result is output to a file where the file name is the input file name with an underscore appended to the front. For example, test.txt as an input file would result in an output file called \_test.txt. The output file contains the code for the recognized characters in the text file.

Option 5: This is the same as option 4, but it decodes a file that was encoded using option 4. This option works the same as option 4 with the naming of the new file.

\*\*Warning\*\* When encoding files, new lines and tabs are stripped from the file and are not replaced when the file is decoded (spaces are fine). All of the characters are in the file, but the formatting of the file is messed up. We were not able to find a solution to the loss of data here.

Option 6: This option takes the name of an input file like option 4 or 5 and then uses the data in the file to train the system. The characters in the file are used to update the frequencies of all of the encodable characters. After the file is read, a new Huffman tree is created and traversed to update the codes.

\*\*Warning\*\* If you update the character frequencies, you may not get the desired result from decoding a file that was encoded using different frequencies. There is a possibility things will be fine, but it is better to re-encode any files after changing the frequencies of the characters.