CISC/CMPE-365
Fall 2019

Week 7 AND Week 8 Lab Problem: Huffman Coding

This is Assignment 3

Due Date, November 2, 11:59 PM

This 2-week lab deals with the Huffman Coding Algorithm for data compression.

Your funding application was 100% successful and the selected projects are all underway. Unfortunately they are generating a huge amount of intranet traffic and since your not-for-profit organization is operating on an antiquated network platform (think dial-up modems) it has become imperative to minimize the size of the information flow.

Therefore you have been tasked with writing a Huffman Encode/Decode package. Here are the specifications:

Bit Simulation:

Huffman Coding works by representing each character as a sequence of bits, constructed one bit at a time. That's a bit challenging (ha ha) in some languages, so for the purpose of this assignment I would like you to simulate bits with "0" and "1" characters. This means that the encoded files will actually be much longer than the originals! But if we remember that each character in an encoded file would actually be a single bit in a "real" implementation of Huffman Coding, then we can evaluate the effectiveness of the coding algorithm.

If you are comfortable working with bits directly, I encourage you to go the extra step and implement the actual bit operations for your own satisfaction. But please hand in the character-level simulation.

Letter Frequencies:

Rather than create a new Huffman Code for each document, it has been decided to base letter frequencies on a canonical collection of documents. This code will then be used for all documents. This has the advantage that the encoding/decoding of each document can be done in a standard fashion. It has the disadvantage of using a possibly non-optimal code for some documents.

Code-Building Module:

The code-building module is responsible for generating code strings for all printable ASCII characters (positions 32 to 126 in the ASCII sequence), plus the line-feed character (which is #10 in the sequence). We will call this set of characters **"the printables"**. The full ASCII sequence can be seen here: https://theasciicode.com.ar/

This module must:

- read all the text files in the canonical collection and determine the overall frequencies of all printable characters and line-feeds. Printable characters that do not occur in the canonical collection should be assigned a frequency of 0.
- apply the Huffman Algorithm to compute code-strings for all characters in **the printables.**
- output the code-strings to a text-file that can be accessed by the other modules. This file should consist of 96 lines, each of which contains an integer representing the ASCII position of one printable character, followed by one or more spaces, followed by the code-string – for which you should use "0" and "1" characters, as explained above.

For example, the first few lines of this output file might look like this:

10 1001
32 101
33 01101110
etc.

Encoding Module:

This module is responsible for encoding a text file. It must:

- obtain the name of a .txt file from the user
- load the code-string dictionary produced by the Code-Building Module
- produce an encoded version of the given .txt file, in which each printable character (including line-feed) is replaced by the appropriate sequence of "0" and "1" characters

Decoding Module:

This module is responsible for decoding an encoded text file: It must

- obtain the name of an encoded text file from the user
- load the code-string dictionary produced by the Code-Building Module
- use the code-string dictionary to decode the file. Here you can exercise some creativity. Your program needs to efficiently determine when the current sequence of (simulated) bits being read from the encoded file constitutes one of the code-strings. One popular solution is to use the code-string dictionary to build a binary tree to represent the entire set of code-strings, then traverse the tree from the root downward, branching left or right depending on the "bit" just read, until a leaf is reached – the decoded character can be stored in the leaf. This way each "bit" of the encoded file is only processed once. Feel free to experiment with other approaches.
- produce a decoded version of the encoded file. This should be identical to the original text file before it was encoded!

Note: I have tried to make sure that all of the text files contain only characters that belong to **the printables**. If any non-printable characters slipped past me, you can ignore them.

Part 1:

Implement the three modules outlined above. Demonstrate their correctness using the file File1.txt to build the code-string dictionary, then encode and decode the file File2.txt to confirm that the decoded version is identical to the original.

In this Part you should show the code-string dictionary that your Code-Building Module generates.

Part 2:

Conduct experiments to explore the effect of using different Canonical Collections.

Canonical Collection 1.zip contains a single file, which is just a list of English words, one per line.
Canonical Collection 2.zip contains several very short documents.
Canonical Collection 3.zip contains a smaller number of quite large documents.

For each Canonical Collection, build the code-string dictionary and use it to encode all the documents in Data.zip. Compare the original size of the documents (measured in bytes) with the "encoded size" (in which each byte represents a single bit).

Determine whether the choice of the Canonical Collection affects the efficiency of the Huffman Coding. If it does make a difference, which of the Canonical Collections gives better results? Can you give an explanation?

Deliverables:

- All of your source code, appropriately documented.
- The code-string dictionary generated during Part 1.
- The results of your experiments conducted in Part 2, summarizing the effectiveness of Huffman Coding for compressing the file sizes when using the different Canonical Collections.
- Your conclusions regarding the three Canonical Collections.