Nick Jones
CS 201
Josh Davis

Huffman Coding

Brief Theory:

The Huffman algorithm for data compression works on a simple premise - the more a character shows up, the less memory we want that character to use. Normal ASCII codes take 8 bits to encode a character. Huffman coding assigns varying length codes to characters depending on how frequently that character appears in the document that is being compressed. The data structure used to organize Huffman codes is a binary tree, with the leaf nodes of this tree corresponding to individual characters. The most important property of Huffman coding is the "prefix principle" - no one code is a prefix of another. In order to maintain the principle of "prefix coding," characters can only appear in the tree as leaves. This prevents any ambiguity when decompressing a file. If a '011010' is seen, we know that this code corresponds to just one character - no prefix of this code (i.e. '01101') or suffix of this code (i.e. '0110101') exists.

Implementation:

Implementing Huffman Trees requires the use of a variety of data structures. The first part of implementing was to create a HuffmanTreeNode class, which was simply a subclass of BinaryTreeNode. This will be discussed in more detail later on. Other data structures utilized included queues, dictionaries, and lists.

The basic algorithms for my program are as follows.

Compression:

- Read through original file, creating Huffman nodes containing a weight (frequency) and value (character) for each character.
- Create two queues; one will be for single nodes (i.e. no children), the other will be for combination nodes (i.e. with children). Enqueue all the nodes just created in the single nodes queue.
- Create a Huffman tree from the bottom up. Take the two nodes of least weight (combined nodes' weights are equal to the sum of their children's weights) and create a new node with these two nodes as the new node's children. Break ties by always taking an individual node when possible. Enqueue the new node in the combination queue, and repeat until one node remains, which will be the root of the Huffman tree.
- Traverse the tree to get codewords. The rule for creating codewords is simple: start at the root, and append a '0' when going to a node's left child and append a '1' when going to a node's right child.
- Store the results of the traversal in a dictionary for fast lookup when compressing the file.
- Read through the original file character by character, replacing each character with its codeword based on the dictionary just created.

- In the compressed file, write out a readable representation of the dictionary of chars => codewords. This is needed for the decompression algorithm to work correctly.
- Then, write out the binary string of codewords - for every eight bits, write out the ASCII equivalent to the compressed file.

Decompression:

- Read the first part of the file, which contains a string representation of the dictionary of char => codewords.
- Read through the second part of the file, converting the ASCII characters back to a string of binary digits.
- Based on the dictionary that was just rebuilt, replace the binary digits with their corresponding characters.

There are three files as part of this implementation: huffman.py, huffmantree.py, and huffmantreenode.py. huffmantreenode.py contains the class HuffmanTreeNode, which is simply a subclass of BinaryTreeNode with the extra instance variable 'weight'. huffmantree.py contains the HuffmanTree class, which has two instance variables: a root HuffmanTreeNode and a dictionary of character => codeword. This file contains the majority of methods needed to compress and decompress files. All of its methods depend on one or both of the class's instance variables. huffman.py deals more with the user interface, and contains functions which act before a HuffmanTree has been created. For example, it contains a function to create the original tree, which can't be done by the HuffmanTree class because it has not yet been created. In addition, it contains the function for reading the first part of the compressed file in order to parse the dictionary needed to decompress the compressed file. Finally, huffman.py contains the large-scale functions compress and decompress.

User Interface:

To use the Huffman coding to actually compress files, the user must pass in a few command line arguments: whether the user wants to compress or decompress, and the name of the file. The following shows the form of command line arguments:

>> python huffman.py [compress/decompress] [filename]

The compress argument creates a new text file; its name is filename + 'Compressed.txt' and displays the size of the original file as well as the size of the compressed file. It does not delete the original file.

The decompress argument creates another new text file: its name is filename + 'Decompressed.txt'. In a real file compression program, the original file would probably be deleted (the point is, after all, to use less memory). Whenever the user needed to fetch the original file again, he or she would simply call the decompression algorithm on the compressed file.
Example: manofthecrowd.txt => manofthecrowdCompressed.txt => manofthecrowdDecompressed.txt

Performance:

In theory, this compression algorithm should act asymptotically. For smaller files, saving the dictionary at the top of the compressed file will take a lot of memory in comparison to the rest of the file. Therefore, as file size increases, we would expect the compression rate to decrease. However, in practice, there's really no need to compress small files - the whole purpose of data compression is to decrease the memory necessary to store very large files. This implementation of Huffman coding reaches its optimal compression rate pretty quickly. Please refer to the appendix for actual data and graphics. The performance of this implementation appears to linear – as the size of the original file increases, the size of its compressed version increases at a fixed rate.

The other main concern of a compression program is the time that it takes to compress and decompress files. Using Python's built-in profiler, I gathered some data on the speed of these processes for the same files as used above. The results demonstrate that decompressing takes significantly more time, and seems to be related linearly to the compression time (as well as the size of the file). Please refer to the Appendix for a table of data.

Conclusion:

This project allowed me to apply a great deal of the knowledge acquired through CS 201 to the real-world problem of data compression. As mentioned above, I utilized a variety of the data structures we discussed in class in my implementation. I also tried to take this project as an opportunity to incorporate abstract concepts (i.e. data structures), low-level memory concerns (i.e. kind of dealing with bits), and user-friendly software design. There are a few things, however, which I would improve upon if given the time.

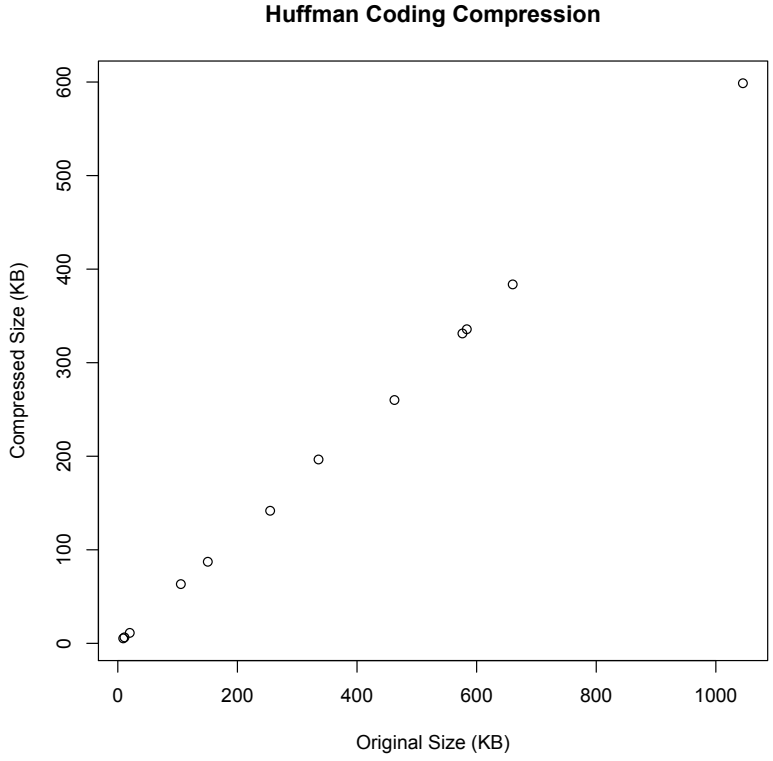
First, my program only compresses text files using ASCII characters. The only problem I had with compressing files throughout the process occurred when trying to do so to a file with other kinds of characters. I didn't manage to work this problem out fully, and as a result my program only deals with ASCII text files. Furthermore, I would spend more time on error-handling. Right now, my program will display error messages when it can't find the files passed in by the users. However, it does not deal so well with trying to decompress files that were not compressed using my program.

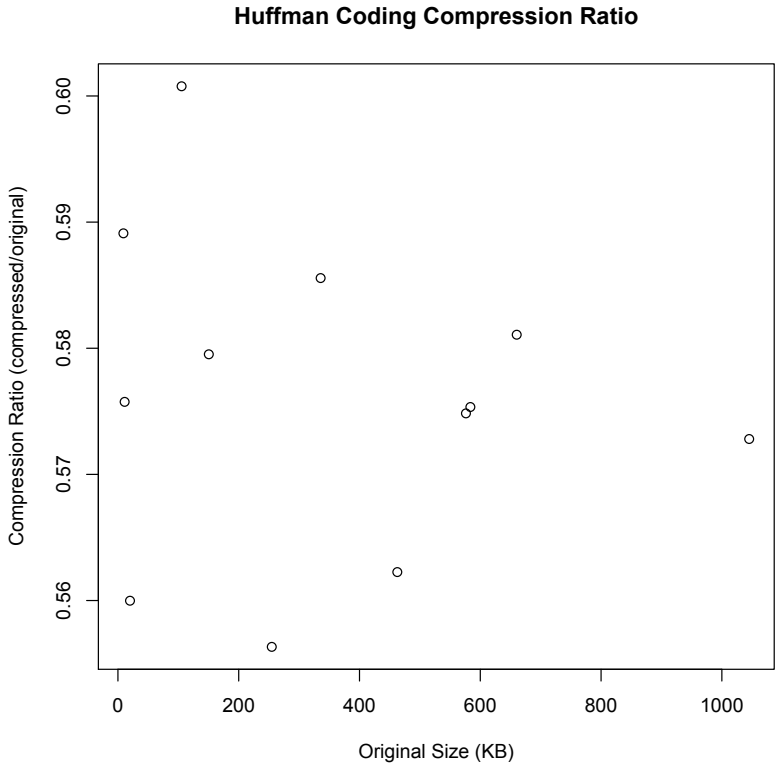
Finally, if given a lot more time, it would be fascinating to delve deeper into how and why some files are more compressible than others (due to factors apart from size). From the few files I compressed in my testing, it looked as though the compression rate was approaching a constant rate, with most of the variability due to other factors. So why do some of them have greater or lesser compression rates? Clearly, the distribution of characters is what affects the rate, but what kinds of texts are more easily compressed, and which are tougher to compress? The files I compressed were arbitrary, and mostly just literature I had heard of or short papers that sounded interesting. Looking into the distribution of characters in varying types of files would be another very interesting project.

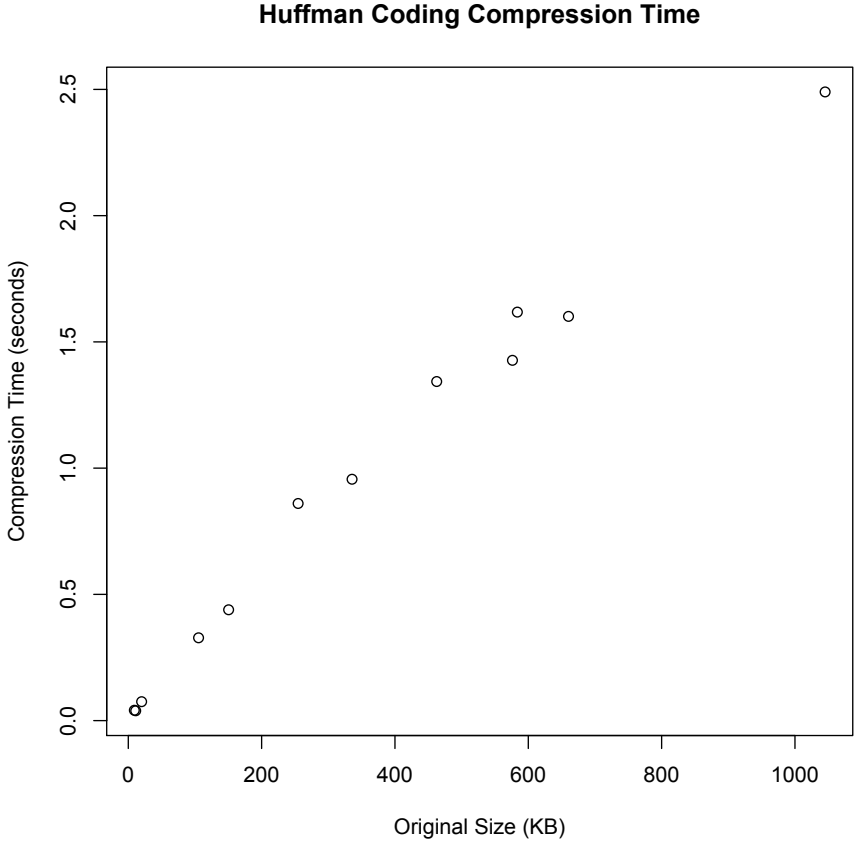
Appendix:

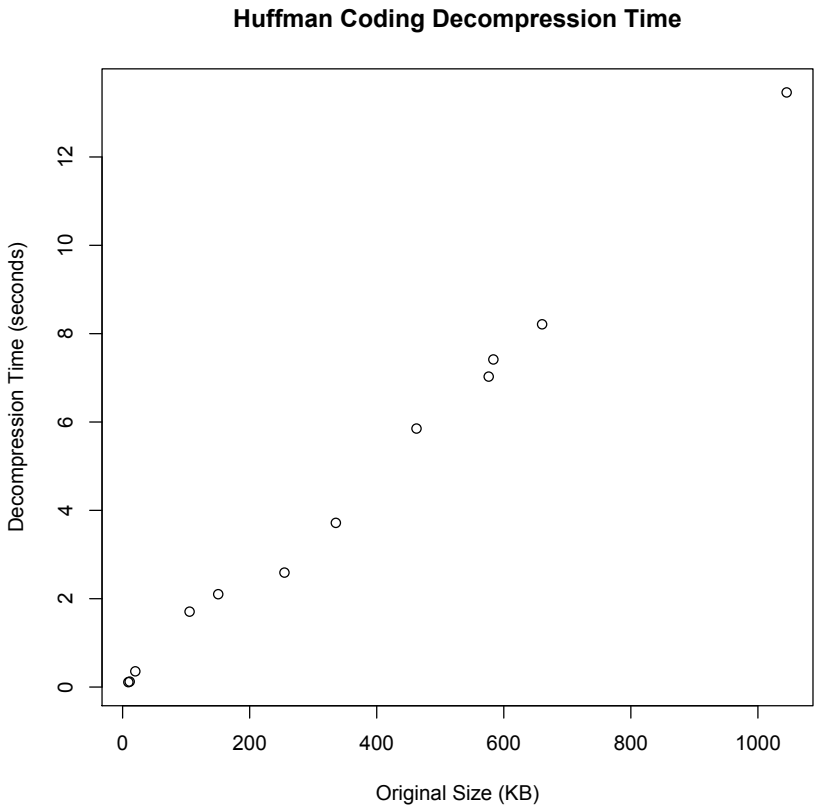
Filename	Original Size (KB)	Compressed Size (KB)	Compression Ratio	Compression Time (seconds)	Decompression Time (seconds)
declaration2.txt	9.057617	5.335938	0.5891105	0.041	0.111
telltaleheart.txt	11.106445	6.394531	0.5757496	0.039	0.123
manofthecrowd.txt	19.992188	11.195312	0.5599844	0.075	0.356
briefhistoryinternet.txt	105.410156	63.327148	0.6007689	0.328	1.707
jekyllhyde.txt	150.493164	87.213867	0.5795205	0.439	2.102
utopia.txt	254.740234	141.71875	0.5563265	0.86	2.591
artofwar.txt	335.639648	196.538086	0.5855628	0.956	3.717
thetrial.txt	462.598633	260.098633	0.5622555	1.343	5.852
sherlock.txt	576.2407	331.248047	0.5748368	1.427	7.028
huckfinn.txt	583.580078	335.757812	0.5753415	1.618	7.416
myths.txt	660.31543	383.691406	0.5810729	1.601	8.212
janeeyre.txt	1045.238281	598.717773	0.5728051	2.49	13.461

Note: These were all taken from a few sites on the Internet; most of them were downloaded from Project Gutenberg, an awesome free ebook site for books with expired copyrights.




Huffman Coding Compression Ratio
Compression Ratio (compressed/original)
0.56
0.57
0.58
0.59
0.60
0
200
400
600
800
1000
Original Size (KB)


Huffman Coding Compression Time
Compression Time (seconds)
0.0
0.5
1.0
1.5
2.0
2.5
0
200
400
600
800
1000
Original Size (KB)


Huffman Coding Decompression Time
Decompression Time (seconds)
0
2
4
6
8
10
12
0
200
400
600
800
1000
Original Size (KB)