Huffman Coding

12/02/2015

Using the code

Note: Code is in Python, version 2.7

Main.py contains the method *run()* which takes a given filename and reads, encodes, and decodes the associated text file. To run the program, open and run *main.py* in Idle on a networked machine.

The code will prompt the user to enter a string corresponding to a .txt file to be decoded. This file must be in the same directory as the program *.py* files. An example file *lorum_ipsum.txt* has been included in the directory for demonstration purposes.

The Code

Main.py is the main file which takes input and calls the relevant encoding and decoding methods from external python files *encoder.py* and *decoder.py*. I/O handling is done through *huffman_tools.py*. The main method also uses *datetime* to evaluate the run time.

The text is encoded by calling *encode_string()* from *encoder.py,* which is passed the given text file as a single string (processed in *huffman_tools.py)* in *main.py*. Firstly a dictionary is generated containing each character that appears in the text and the frequency with which they appear. This dictionary is then copied into *tree* and used to build a huffman tree by combining the two smallest frequency elements over and over.

Once the tree is completed, *walk_tree()* is then called which passes through the tree in order to generate the unique binary key for each character – the crux of the huffman coding technique. With this binary dictionary, the string is converted into first binary, which is then padded and turned into bytes in *string_to_bits()*. The number of bits used to pad the byte array is added as an initial byte. This byte array is concatenated with the dictionary (which is needed for decoding) and saved to the appropriate output file (*<input filename>.hc*).

Decoding simply reverses this process. The compressed file is read and passed to *decode_string()* where the dictionary is sliced off the front of the input and recreated, the padding size (*bufferCount*) found, and the remaining bytes converted into a string of 1/0s. The padding bits are removed and this string is converted into the correct characters again using the dictionary. Because the tree creates unique binary keys for each character and no key contains another as a substring, there is no risk of mistranslating.

Results

To test the compression methods efficacy it was run on a number of input text files of increasing size. As can be seen, at file sizes of ~1kB, little compression is achieved, and in fact at even smaller file sizes, the compressed file size is actually larger than the input size. This is because of the size of the binary dictionary needed to decode the file, which is consistently of a certain size. However, as the input size increases, the added size of this binary dictionary quickly becomes irrelevant, and the compression ratio stabilises at ~55%.

This compression ratio is affected by two key factors: the variety of characters, which expands the dictionary and means that more binary keys are needed, of a longer length; the frequencies of these characters, as this distribution can lead to big skews in binary key lengths.

File Size (kB)	**Compressed Size (kB)**	**Compressed %**
1202	1184	98.5
6263	3994	63.8
12366	7262	58.7
25829	14453	56.0
49281	26994	54.8
91084	49348	54.2
3291648	1877097	57.0

The run time of the program was good, as can be seen below. In terms of time complexity performance, the growth is O(kn), where k is constant.

File Size (kB)	**Encoding Time (s)**	**Decoding Time (s)**	**Total Time (s)**
1202	0.0018	0.01	0.0118
6263	0.034	0.02	0.054
12366	0.031	0.04	0.071
25829	0.052	0.08	0.132
49281	0.059	0.15	0.209
91084	0.099	0.29	0.389
3291648	2.88	11.15	14.03

Discussion

Overall, I am satisfied with my Huffman coding implementation. It works, achieving a reasonable level of data compression (~55%) and does so pretty quickly.

I played about with implementing Lempel-Ziv-Welch compression in addition to Huffman but unfortunately was unable to complete this implementation in time. However, this would have (particularly for larger files) have notably increased compression.

In terms of implementation, there were a few smaller improvements that could have been made. For simplicity's sake I chose not to write the dictionary to bytes, which would have made the dictionary a little smaller. Finally, passing the text file as a single string in one go is technically inefficient memory wise, though this makes no impact at these small input sizes. For handling much larger text files, the characters could be read one by one.