Huffman Codec Report

Instructions

My Huffman encoder and decoder is written in Java 7 and packaged in the form of an encoder.jar and decoder.jar.

The encoder can be used from the command line like:
java -jar encoder.jar [path to file to be encoded] [number of bytes per block] [path to write out to] e.g.
java -jar encoder.jar lorem.txt 1 lorem.hc

The decoder can be used similarly from the command line like:
java -jar decoder.jar [path to encoded file] [path to write out to] e.g.
java -jar decoder.jar lorem.hc decoded_lorem.txt

The encoder and decoder do not work on the Vega Linux service because it using an outdated version of Java (Java 6).

Program Design

Bytes vs Characters

I decided to use blocks of bytes as the units of encoding, rather than characters. I attempted to code my program so that the Huffman encoding can be done e.g. one byte at a time, or two bytes at a time, etc. This means that my encoder would work for fixed-length text encodings (i.e. ones where every character is encoded as the same number of bytes).

There is a bug in my code though where although it will always work for blocks of one byte length, it will sometimes fail for block lengths of more than one byte. Therefore my encoder works for ASCII text files and other text encodings where there is an equivalence between bytes and characters.

Bitstream

For working with bitstreams, I created two streams which transform between ASCII '1's and '0's and bitstreams.

- ASCIIToBitstreamWriter - instances accept an output stream in their constructor. Strings of ASCII 1s and 0s can be written to an instance and it will write out 1 and 0 bits to its output stream. When this instance is closed, it writes out 0 bits to its output stream so that the total number of bits written out is a multiple of 8 (i.e. whole bytes).
- BitstreamToASCIIReader - instances accept an input stream in their constructor. The read(int n) method returns n bits from the input stream in the form of an ASCII string of 1s and 0s.

These classes use a helper class, ASCIIBitstring, which just acts as a bidirectional map between all 256 8-bit bitstrings and their unsigned integer representation. In my program I mainly work with ASCII 1s and 0s rather than bytes or bits, and then let these bitstream classes deal with converting any ASCII 1s and 0s strings to bits on disk and visa versa.

.hc file format

The first byte is interpreted as an 8-bit natural number, and indicates the size of the blocks in bytes. The second, third, fourth and fifth byte are collectively interpreted as a 32-bit natural number, indicating the number of blocks for which there are encodings. This information is necessary for when rebuilding the Huffman tree. There is then a representation of the Huffman tree in terms of 1s and 0s. For storing Huffman trees, I used the method as described in reference [1]: "The easiest way to output the huffman tree itself is to, starting at the root, dump first the left hand side then the right hand side. For each node you output a 0, for each leaf you output a 1 followed by N bits representing the value." . The length of this representation (in bits) is (the number of leaves - 1) 0-bits (representing the branches), (number of leaves) 1-bits (representing the leaves) and then the (block size * number of leaves) bits which represent the actual bytes of the blocks themselves. Straight after the Huffman tree comes the actual encoding of the text as according to the Huffman tree.

Testing

I tested my implementation on four different text files (as in the test/ directory). Pre-generated results are in test/premade_results/ directory).

- lorem.txt - some generated lorem ipsum text from lipsum.com
- pi1000000.txt - million digits of pi
- test.txt - a string of b's, c's, d's, e's, f's and newline characters such that the frequencies match the frequencies as in reference [1]'s example (although reference [1]'s example uses the digits 1-6
- war_and_peace.txt - one of the largest plain text files available from Project Gutenberg

Analysis

Compression Ratio

// TODO: table

My implementation seems to compress to around 50% of the original file size for text files.

Running Time of Encoder

Counting Blocks

The encoder first does one pass over the input file in order to find out what blocks appear in the file and how many times each block appears. It does this using the BlockReader class, which accepts an input stream of bytes (i.e. some file) and outputs blocks of fixed length (e.g. blocks of 1 byte length, blocks of 2 byte length, etc). I also created a Counter class to help with counting. The Counter class takes the BlockReader and reads Blocks from it, keeping count as it goes along, until the stream is exhausted. After this step is complete, the counter contains both the set of all blocks that appear in the file and also their relevant counts.

With regards to running time, for a file of n bytes, there are n read() calls to the file input (as only one byte is read at a time from the file), which takes $\Theta(n)$ time. Assuming the construction of block instances from the raw bytes some constant amount of time ($\Theta(1)$), and the counter instances internal get(), put() and containsKey() calls are constant time as well (as the counter uses an internal HashMap to keep track of counts), then roughly this step takes time proportional to the length of the file in bytes.

Building the Huffman tree

Next the Huffman tree is created (re: HuffmanTree.fromBlockCounter()). First a PriorityQueue is created which can hold nodes of a Huffman tree, and to it all the encountered blocks are added as dummy HuffmanLeaf instances (Huffman leaves encapsulate a block and the count of that block). The PriorityQueue maintains as the topmost node the one which has the least count. The method as described in the lectures (and also in reference [1]) is used. Two nodes are popped from the priority queue at a time and paired together under a branch node (a HuffmanTree instance) and this is done until there is just one node left in the PriorityQueue; this last node is the root of the Huffman tree.

Adding nodes to a priority queue of size n takes O(log n) time.[2] For n blocks then, adding them all to the priority queue takes O(n log n) time. After this, the while (2 <= nodes.size()) loop reduces the size of the priority queue by one each time, so the loop will run for n - 2 times i.e. $\Theta(n)$ times. Then the root node can be popped from the priority queue. Overall this step takes O(n log n) time.

Encoding

Next the encodings for each block are generated and stored in a HashMap (re: EncodingBuilder.getEncodings()). The creation of this map requires traversing all nodes in the Huffman tree, which for n blocks is (n - 1) branches and (n) nodes, i.e. 2n - 1 nodes must be traversed. For each branch reached, there is a string concatenation operation (going left prepends a 0 to a code and going right prepends a 1), so n-1 string concatenation operations. For each leaf reached, there is a put() operation to the map, so n put() operations. Assuming constant time for checking whether a node is a branch or a leaf, for string concatenation and for putting a key into the HashMap, this step should take time proportional to n.

Writing out the .hc file

First the number of bytes per bloc and the number of blocks are written out to the file. Then a string representation of the Huffman tree is constructed and written out. Then the input file is opened up again and a pass is done over it, converting it to blocks of the correct size and

Running Time of Decoder

First the number of bytes per block and the number of blocks are read in and parsed from the first five bytes of the file. Next the Huffman tree must be reconstructed from its bitstring representation.

Limitations

As the codec works byte-wise, it will only meaningfully encode text files which use a fixed-width text encoding, e.g. ASCII or UTF-32. For example, ASCII characters are all one byte long, so an ASCII text file can be meaningfully encoded by setting the block size of the Huffman encoder to 1. For UTF-32, every character takes up exactly four bytes, so UTF-32 text files can be meaningfully encoded using a block size of 4. Variable-width text files e.g. UTF-8, where each character may be between 1 and 4 bytes in length, can still be encoded but the leaves of the Huffman tree will not necessarily correspond to characters. This limitation of the codec could be circumvented by converting any variable-width text file to a fixed-width text encoding before Huffman encoding.

My implementation is untested for encoding empty files and encoding files where there is only one block present, my implementation does not handle inputs of this type.

Bugs

There are large but finite size limits on various parameters of the Huffman encoding. The maximum block size is 255 bytes. There is also a finite limit on the number of different blocks that can be encoded in one file - specifically Integer.MAX_VALUE of Java, which is 2**31 - 1.

Possible Improvements

Below are some improvements I would have liked to have made if I had more time available for this project.

I could have used traversal of the tree to do decoding rather than building a HashMap of ASCII 1s and 0s.

I should have found a way to delimit the encoded stream so that the decoder doesn't think the end padded 0s of a .hc file are encodings. One way I could have done this would have been to precalculate how many bits the encoded stream would take up and then put this number in the header of the .hc file and have the decoder stop after reading that many bits.

The input file should be locked for the duration of the process (currently it is released after all the blocks have been counted, and is not opened/locked again until after the Huffman tree has been built). This is because if the input file is altered in the time between the first pass (counting the blocks) and the second pass (encoding the blocks and writing the out simultaneously to a .hc file), then the program will fail.

I should have created buffer clasess for the ASCIIToBitstreamWriter and BitstreamToASCIIReader stream classes, as currently they read 1 byte from an input stream at a time. For files this is very inefficient as on some operating systems this will require a system call for every single byte written which adds overhead to the program. A better solution would have been to use a buffer class which say maybe read in 1KiB of bytes at a time from the input stream stored them in a buffer.

References

1. https://www.siggraph.org/education/materials/HyperGraph/video/mpeg/mpegfaq/huffman_tutorial.html
2. http://algs4.cs.princeton.edu/24pq/