Practical 1: Adaptive Huffman Coding

Introduction:

Originally proposed in 1952 by David Huffman while working on his Ph.D., Huffman Coding is a well know method of lossless data compression. His original proposal was based on the idea of reading a source file and calculating the probabilities of a certain symbol appearing in one pass, then building up a tree using these probabilities. This tree is then used on a second pass of the file to encode the symbols seen. This is a somewhat involved process though as the file needs to be scanned twice in static Huffman coding. However, algorithms were developed - the most popular of which are the FGK and Vitter [2] (Vitter being an improvement on FGK) – which allow the Huffman codes to be developed on the fly with 1 pass through a source file. This works by updating the code tree every time a symbol is seen, and adjusting the locations of nodes within the tree so that they follow a set of invariants. These are:

(1) A node with a higher weight has a higher index.

(2) Sibling nodes have consecutive indices.

(3) Internal nodes have higher indices than their children.

Design (with extension discussion of algorithm):

The algorithm used to produce the practical was essentially that which was provided by Vitter [2] as laid out in [3]. Initially, we start with an empty tree, which consists of only a root node which is also the NYT node (not yet transmitted). The first byte of the input source file is then read. As nothing has been seen before, we simply output this byte to the output source and then the byte is added to the tree. In the tree, the byte is represented as an integer number (but more on this in the implementation section). On insertion into the tree of a symbol, the NYT node “gives birth” to a new node that contains the symbol seen, and also to a new NYT node. This old NYT node then becomes an internal node in the tree. The key to Vitter’s algorithm is that on insertion of a new node into the tree, or if a symbol is seen again and its weight must be incremented, then the tree is searched to find all nodes that have the same weight as the node to be incremented. If a node exists with a higher index and same weight (by sibling property adjacent nodes have consecutive indices, so siblings of same weight still have increasing indices), then the node to be incremented is swapped with highest indexed

node. The swap process leaves the indices as they were though, and a swapped node brings all children with it.

After the nodes are swapped, the weight is incremented and the process propagates up the tree to the parent of the swapped node. This continues all the way to the root, swapping nodes as necessary to retain the sibling property. However, a node must never be swapped with its parent, and the root is also never swapped with anything.

Some limitations that arise from the Vitter method are speed related. Due to the fact that the tree needs to be searched to find nodes with the same weight during the update process, there is a speed penalty involved. However, in the implementation given this is combatted by using a list of nodes that are kept in index order. Due to the property that siblings have increasing indices, finding nodes of the same weight only involves searching through the Order list from the current node's index until we find the highest index with the same weight – which is usually a short process.

A diagram of the Vitter Adaptive Huffman Update Procedure is given at the end in Images section. [3]

Implementation:

To implement this adaptive Huffman encoder and decoder, first a tree structure was implemented. Initially the plan was to have an abstract class of *Node* which had pointers to left and right children, as well as parent nodes. Subclasses of type *NYTNode*, *LeafNode* and *InternalNode* would then be created to represent all possible states of a node in the tree. This was however a rather complex plan which required a lot of *instanceof* checking. Thus it was decided that instead there would only be one *Node* class. It still contains pointers to left and right children and parent, but now also contains private fields for index, value (if it is a leaf node) and Boolean flags to determine if the node is NYT or a leaf. This was a much simpler implementation and allowed for some cleaner code. Having pointers to both children and parent was also a useful design decision because it allows for very easy traversal of the tree in both directions.

The tree itself consists of a pointer to the root node and the current NYT, as well as 2 private fields which are a map of integer values to nodes in the tree, and a list of nodes in the tree in

index order. The map of ints to nodes allows to easily check whether a value exists in the tree in near O(1) time (depending on how busy the map is). For ease of indexing, an arraylist of nodes is used, called *order*. Whenever a new node is inserted into the tree, it is inserted at the front of the lost (index 0), so that the indices of the nodes already in the tree are updated correctly. Likewise, whenever a swap occurs, it is simply a matter of setting the corresponding locations in the list to the correct nodes. One slight implementation detail that affects speed is that the nodes all contain an integer field with their index in it, this field is not strictly necessary for anything but it makes printing a node easier as its index can be output. Due to this, whenever nodes are inserted into the tree, every node in the *order* list must be updated so that its internal reference to its index is correct. This slows down tree operations as the entire list needs iterated over, and could be removed to improve speed, but it was left in as being able to see node indices on printing was very useful during testing of the tree.

The actual tree operations follow that of Vitter's algorithm. The tree starts out as an NYT node, and on seeing first symbol, the NYT uses the *giveBirth* method to create a new leaf and NYT node. Then the *updateTree* method is used to maintain the sibling property. Structure for the update process was in part borrowed from the pseudo-code layout of Huffman Encoding provided Stringology.org [4]. However, because of the order list, finding the highest indexed node in a weight class is a fairly trivial operation (strictly speaking it is O(n) but very small n, especially for small trees) and so updates don't take too long. After determining if a node needing to be incremented is the highest index in its weight, a swap is performed. Swapping is again a fairly simple operation, which just requires changing the pointers to parent nodes, and the pointers to children in parent nodes, indices are also sorted out so that the new node being swapped receives the correct index.

For code generation in the encoder, the tree is traversed from leaf (or NYT) to root. This is because it is simpler to fetch a leaf node from the *seen* map and traverse up the tree using the *parent* field than it is to search from root down. To actually generate the code, a list of Booleans is built up. Each time we move to a parent node, we check if the previous node was the right or left child, and if right then add TRUE to the Boolean list otherwise we add false. This list of Booleans then represents the code for the corresponding value but in reverse order, so we simple read the list backwards and output each bit to the output file as we encounter it. Bit output is handled by the *BitByteOutputStream* class which is a wrapper around a *FileOutputStream* object. It contains a buffer of bits to write to the output stream and flushes the buffer to output when it contains 8 bits.

For the decoder, the process is slightly simpler. First, when the tree is empty, we read the first byte from the file and insert it into the tree. From this stage on, we read one bit at a time, and traverse the tree from the root, going left for 0, right for 1. If the code corresponds to the NYT node, then another byte is read in and inserted into the tree and tree updated. If the code corresponds to a leaf, then it is incremented and the tree updated. Code for the single bit input stream in ***BitInputStream*** class was taken from a project on GitHub.com [5].

Testing:

To test the initial setup of the tree, some unit tests were made. These test just check that the tree inserts nodes properly and that the indices are updated correctly. Thanks to the testing, some problems were noticed with the initial code for tree manipulation. Initially, the swap method was not updating the node.parent pointers when swapping nodes, which meant that even though the new parent nodes knew their children, the child did not know its parent which broke tree traversal from leaf to root.

For testing the encoder and decoder, text files of varying length were used and a good compression ratio of 1.7 or above was achieved (see images section for output screenshots). Also tested was a large media file (an mp3 of roughly 13MB), and the compression ratio was a little disappointing (still more than 1) but at least it decoded perfectly. Some tweaking of the code will probably need to be done for audio files though, as well as images, to read in more than just a byte at a time.

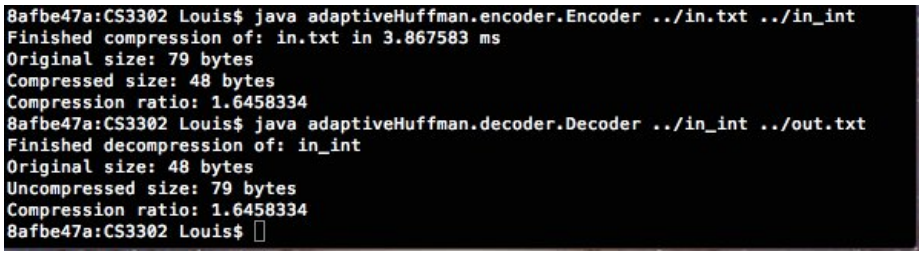
Outcome:

In general, the outcome of the practical was pleasing. The encoder and decoder work well on text, generating good compression ratios, and compression does not take an overly long time. As highlighted above, performance could improve by removing the unnecessary index values in nodes, which would allow the O(n) index update method to be dropped. It will be left in for now though to make printing easy. However, the rest of the implementation is solid, with easy methods to check existence of symbols in the code tree, and robust code generation.

If time had permitted, further experimentation would have been done on varying file types, by allowing the tree to read in more than just a fixed 8-bit byte. Also, comparison with other lossless compression methods would have been a pleasing addition.

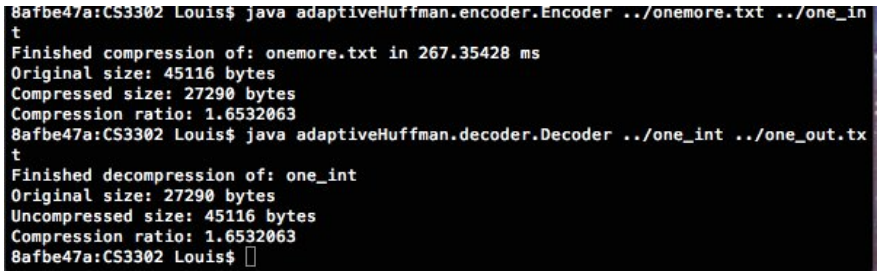
Experimentation:

The first experiment conducted was on a small (<80 bytes) file of plain ASCII text.
The original file was 79 bytes. After compression, the output was 48 bytes, with a compression ratio of 1.646 achieved in 3.8milliseconds. The input, intermediate and output text are included in the archive.



Output from small test file.

Next, a larger body of text was used.
It was 45 KB to start. After compression it was 27KB with a compression ratio of 1.653 achieved in 267.4ms.



Output from medium test file.

Test 3:
Input: 93KB
Output: 55KB
Ratio: 1.67
Time: 405ms

```
8afbe47a:CS3302 Louis$ java adaptiveHuffman.encoder.Encoder ../mids.txt ../mid
Finished compression of: mids.txt in 404.9954 ms
Original size: 92617 bytes
Compressed size: 55468 bytes
Compression ratio: 1.6697375
8afbe47a:CS3302 Louis$ java adaptiveHuffman.decoder.Decoder ../mid ../mid_out.txt
Finished decompression of: mid
Original size: 55468 bytes
Uncompressed size: 92618 bytes
Compression ratio: 1.6697556
8afbe47a:CS3302 Louis$
```

Output from large test file.

Test 4:

Input: 320KB

Output: 179KB

Ratio: 1.783

Time: 1093ms

```
8afbe47a:CS3302 Louis$ java adaptiveHuffman.encoder.Encoder ../hotb.txt ../hotb_int
Finished compression of: hotb.txt in 1093.5237 ms
Original size: 320006 bytes
Compressed size: 179447 bytes
Compression ratio: 1.7832898
8afbe47a:CS3302 Louis$ java adaptiveHuffman.decoder.Decoder ../hotb_int ../hotb_out.
txt
Finished decompression of: hotb_int
Original size: 179447 bytes
Uncompressed size: 320007 bytes
Compression ratio: 1.7832954
8afbe47a:CS3302 Louis$
```

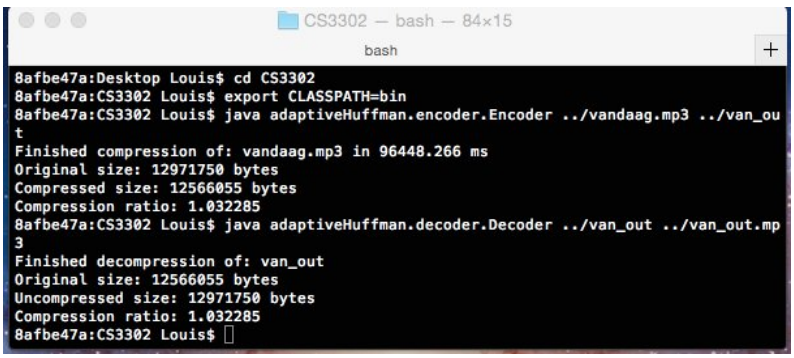
Test 5 – 13MB mp3 File vandaag.mp3

Input: 13MB

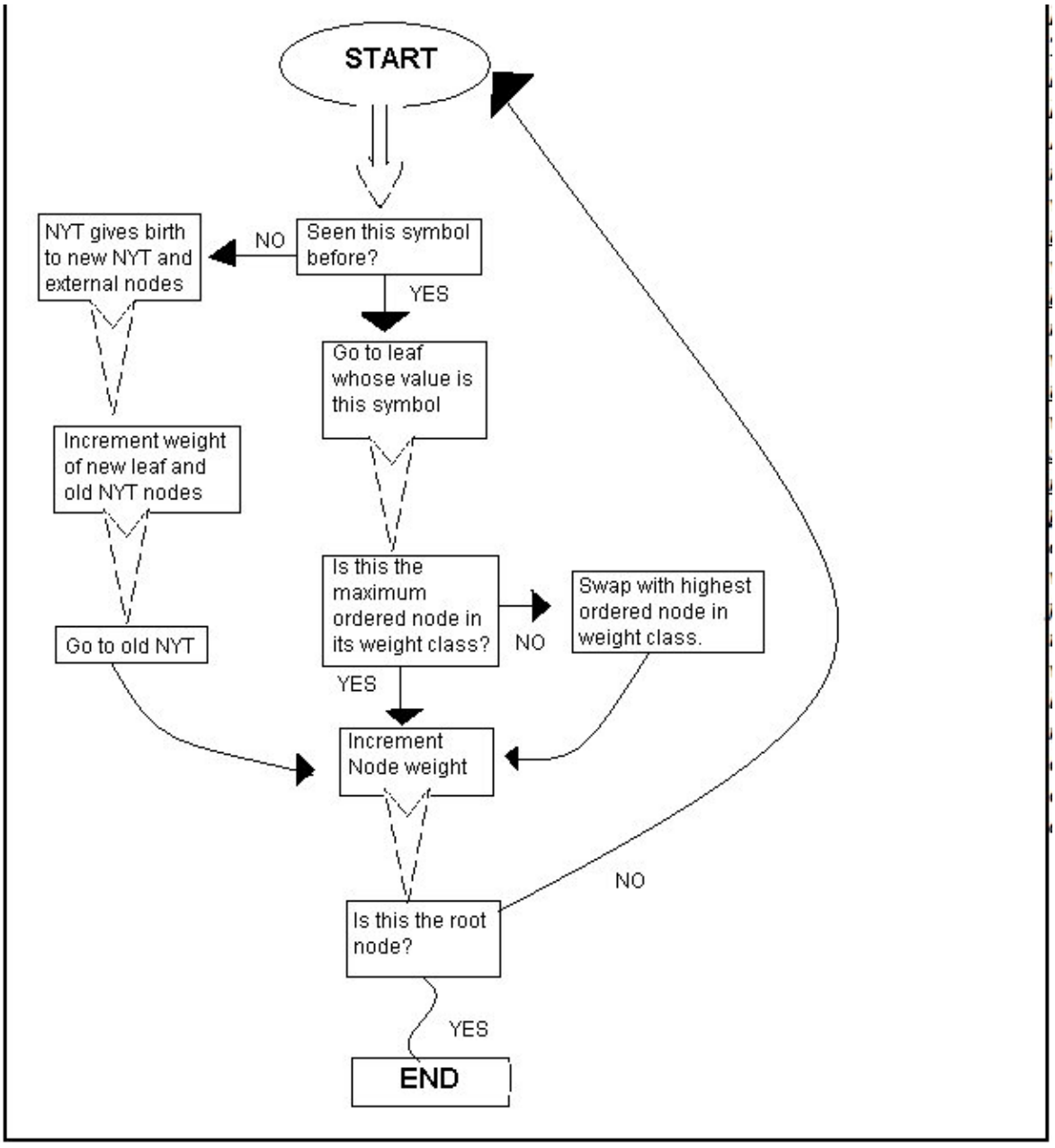
Output: 12.6MB

Ratio: 1.03

Time: 96.4s



Images:



Vitter's Huffman Tree Update Procedure [3].

References:

[1]: Huffman, D., "*A Method for the Construction of Minimum-Redundancy Codes*", Proceedings of the IRE, 1952, http://compression.ru/download/articles/huff/huffman_1952_minimum-redundancy-codes.pdf

[2]: Vitter, J., '*Algorithm 673 Adaptive Huffman Coding*', ACM Transactions on Mathematics, 1989, http://www.ittc.ku.edu/~jsv/Papers/Vit89.algojournalACMversion.pdf

[3]: Low, J., '*Adaptive Huffman Coding*', Image taken from webpage: https://www.cs.duke.edu/csed/curious/compression/adaptivehuff.html

[4]: Adaptive Huffman Coding pseudo-code http://www.stringology.org/DataCompression/fgk/index_en.html

[5]: Class BitInputStream, from user Nayuki on GitHub, https://github.com/nayuki/Arithmetic-Coding/blob/master/src/nayuki/arithcode/BitOutputStream.java