Arithmetic Coding:

Arithmetic coding was first proposed by C. Shannon in 1948 as a form of lossless compression [3]. Since its implementation (by Risannen and Pasco in 1976) it has become the industry standard particularly for JPEG, LZMA, and BZIP [5]. It took a few decades after the first implementation for arithmetic coding to become popular, due to an aggressive patent enforced by IBM that only elapsed after a couple of decades. However, arithmetic coding was so impressive that it resulted in a separate piece of code, called range coding, which was invented in 1979 and performed in the same way but was free from patents [4].

The reason it was so highly regarded was due to how close the algorithm gets to entropy; it is close to the best possible compression rate for a particular statistical model. The algorithm allows the whole file to be expressed as a singular number between 0.0 and 1.0. This seems quite miraculous; however, it uses key concepts from binary search to implement this compression. Each character is given a statistical probability, which then allows the interval of [0, 1) to be divided into subintervals, each representing that character. This process is repeated, with the subinterval corresponding to the input character chosen for the next iteration. A number in the final range is then outputted as the codeword for the entire sequence of symbols. It relies on the fact there is an infinite number of numbers between 0 and 1 so, theoretically, incredibly long pieces of data can still be encoded using this process. Its best cases are with skewed alphabets; the more likely the symbol the less digits the compression will use [2].

Decompressing a file is a similar process to encoding, just in reverse. In short, to decode the text, one must “draw” a vertical line through the subdivided ranges, updating the subinterval whilst moving towards the range the number lies in. The interval containing the codeword is a subset of every interval created in the encoding process. By checking where it lies and outputting that character and then subdividing that interval, the string is reconstructed.

Unfortunately, arithmetic coding is limited by the precision of computers. The biggest drawback of arithmetic coding is how slow it is. It achieves an almost optimal compression rate in conjunction with an adaptive model but at the cost of speed. Therefore, a newer model has been created using approximations of arithmetic coding, which improves the speed of compression with only a slight degradation of compression performance [4].

Unlike Huffman coding, where the probability information must be built into the coding tables, arithmetic coding and modelling are separate so probability estimates can be changed adaptively. This solves the problem of which VLC to use as it doesn’t adversely affect compression rates.

Huffman Coding:

Huffman coding was devised by David Huffman in 1952 [6] the principle was based on 2 statements:

1. 2 symbols with least frequency should have the same length code.
2. Most used characters will have shorter code.

Huffman is ideal for compressing text or program files, which explains why it is used a lot in compression programs like ZIP or ARJ [9]. The time complexity of Huffman encoding is O(nlog(n)) where n is the number of unique characters, which has been proven to be optimum for any method that assigns a fixed encoding to each character [7].

Huffman coding is another type of entropy coding. However, in this instance, it uses knowledge of source statistics to generate a binary tree used to encode characters uniquely based on their statistical likelihood. In essence, Huffman coding works by creating a frequency array of all the characters. It then turns all the characters into leaf nodes. The 2 leaf nodes with the least weight (frequency) are joined together with a new node which is then added back into the list with the combined weight of the 2 previous leaf nodes. This process is repeated until a tree incorporating every node has been created. Next, characters are assigned codes based on their position in the tree, the most frequent will have shorter codes than the less frequent, as they are higher up the tree. All of the text is then converted into code. As a normal ASCII character is made up of 8 bits, there is usually a significant compression by using this alternative method of encoding characters.

As for decoding, the tree that creates the code is required to be used as a “key”, as it goes through the code, reading the characters until it reaches a leaf node, where it then converts that codeword back into the character.

A large part of Huffman encoding is making use of data structures such as the tree that allows the characters to be encoded. An array can be used to store the characters and a hash map to store frequencies, would also improve efficiency. Another data structure that could be used is a min heap that contains nodes.

Since it is a fairly simplistic algorithm, there are many different methods to implement it. The time complexity of Huffman encoding is O(nlog(n)), where n is the number of unique characters.

Like Arithmetic coding, Huffman coding is a greedy algorithm, as it deals with generating minimum length prefix-free binary codes.

Lempel-Ziv-Welch Coding:

Lempel-Ziv-Welch (LZW) coding was invented by Abraham Lempel, Jacob Ziv and Terry Welch in 1977 [11]. Like arithmetic coding it also had a patent from Unisys which made it much less popular post 1990’s. A common theme among compression algorithms and resulted in free versions with very similar logic. It is most typically used in GIF’s, the Unix default compress command and can be used in PDF’s and TIFF’s [10].

LZW uses a dictionary to store groups of previously seen sequences of strings of various lengths and assigning codewords to them. This means it works best on files with a lot of repetition (such as image files). Conversely, if there is no repetition in the file then it can make the file even bigger when encoding it [11].

Put simply, the algorithm is made up of five steps. First, initialise the dictionary containing all strings of length one. Next, find the longest string in the dictionary that matches the current input. Then, emit the dictionary index for that string to output and remove that string from the input. Finally add that string followed by the next symbol in the input to the dictionary. Repeat the process of finding the longest string until the input is empty. For text with a lot of repeated strings this results in a significant compression.

The decoding works by reading through the dictionary and matching codewords with their associated string. The full dictionary is not required, only the initial dictionary containing the single string characters, as it is reconstructed during the decoding process [11]. When the decoder encounters a string that isn’t in the dictionary it breaks it down into 2 pieces, the root string and the appended character It outputs the code for the root string, and adds the root string plus the appended character to the dictionary. Then it starts creating the new group of strings that start with the appended character. The decoder is always one step behind the encoder, which can prove to be problematic. This means if the encoder ever sends the very latest code in its dictionary, the decoder will be unable to decode it [10].

LZW coding is very fast, unlike arithmetic coding, due to its simplicity, making the optimum use cases of the algorithm different. It can compress the input stream in a single pass and requires no prior information about the input stream (unlike Huffman coding). Though, due to modern advancements in computing, all recent computers have the ability to use more efficient algorithms as LZW is fairly old. Like arithmetic coding and Huffman coding, LZW is a greedy algorithm, as it tries to find the longest possible string that it has a code for.

Comparisons:

Huffman and LZW are fairly similar, using codewords to encode the text into binary. The major difference between LZW and Huffman is that LZW is about the frequency of repetitions of strings and Huffman is about the frequency of single characters occurring. LZW is dictionary-based whereas Huffman is binary tree based. However arithmetic coding is different to both of them as it doesn’t use code words rather it encodes the entire message into a single number. Asymptotically, arithmetic coding consists of few arithmetic operations, so its complexity is less than Huffman’s. LZW’s asymptotic complexity is implementation dependent however for large amounts of data it is comparable to Huffman’s algorithm.

[1] M. Pu, “Arithmetic Coding,” *Arithmetic Coding - an overview | ScienceDirect Topics*, 2006. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/B978075066310650009X#!> [Accessed: 05-Mar-2021].

[2] P. Howard and J. Vitter, “Practical Implementations of Arithmetic Coding,” *Gatech*, 1992. [Online]. Available:<https://www.cc.gatech.edu/~jarek/courses/7491/Arithmetic2.pdf> [Accessed: 05-Mar-2021].

[3] G. Langdon, “Introduction into Arithmetic Coding,” *Intro\_Arith\_coding.pdf*, 1983. [Online]. Available: <https://www.cs.cmu.edu/~aarti/Class/10704/Intro_Arith_coding.pdf> [Accessed: 05-Mar-2021].

[4] P. G. Howard and J. S. Vitter, “Arithmetic Coding for Data Compression, 1994; Howard, Vitter,” *SpringerReference*, 1994.

[5] C. McAnlis, Google, 2015.

[6] K. Sayood, “Huffman Coding,” *Introduction to Data Compression (Fifth edition)*, 03-Nov-2017. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/B9780128094747000033.> [Accessed: 05-Mar-2021].

[7] A. Diwan, “Huffman Coding Algorithm,” *studytonight.com*, 2012. [Online]. Available: <https://www.studytonight.com/data-structures/huffman-coding#:~:text=Huffman%20Encoding%2D,to%20the%20given%20problem%20statement.&text=Priority%20Queue%20is%20used%20for,functionality%20of%20a%20priority%20queue.> [Accessed: 05-Mar-2021].

[8] T. Scott, 2017.

[9] M. Nelson, “LZW Data Compression Revisited,” *Mark Nelson*, 08-Nov-2011. [Online]. Available: <https://marknelson.us/posts/2011/11/08/lzw-revisited.html> [Accessed: 05-Mar-2021].

[10] A. R. Saikia, “LZW (Lempel–Ziv–Welch) Compression technique,” *Geeks For Geeks*, 2019. [Online]. Available: [https://www.geeksforgeeks.org/lzw-lempel-ziv-welch-compression-technique/.](https://www.geeksforgeeks.org/lzw-lempel-ziv-welch-compression-technique/) [Accessed: 05-Mar-2021].

[11] S. Says: N. says: J. Says: A. Says: D. S. Says: J. says: L. says: M. U. Says: and G. N. A. N. I. says: “LZW compression: What it is & its use in TIFF & PDF,” *Prepressure*, 25-Aug-2020. [Online]. Available: <https://www.prepressure.com/library/compression-algorithm/lzw.> [Accessed: 05-Mar-2021].

Weekly progress log:

Week 3- Watched a video on Huffman encoding, broke down the algorithm into steps.

Week 4- Researched binary trees and ways to make frequency tables.

Week 5- Watched videos on Arithmetic Coding, LZW and Fixed-Block compression. Decided to do the lit review on Arithmetic Coding and LZW.

Week 7- Found the references for my lit review, began to construct and essay plan and break down the specification. Researched more about Huffman coding, looked at the difference between dynamic and the original Huffman encoding. Began to implement Huffman coding in java, starting with the nodes and getting frequencies of characters.

Week 8- Looked at traversing the tree to get the code. Completely finished lit review and had it checked for grammatical errors.

Week 9- Started on encoding the data and writing and reading from a text file.

Week 10- Made a mini interface and finished the performance analysis.

Design Decisions:

Data structures used:

String: To store the symbols for the tree, the Code for each of the symbols and the names of files.

HashMap: To store the frequencies of the symbols and the nodes they correspond too.

Array List: To store the Nodes to form a tree that can then later be used in encoding and decoding the data.

Binary Tree: Used in Huffman encoding to create codes for each of the symbols.

Algorithms:

Implemented the Huffman algorithm by finding the frequency of characters, then creating a binary tree and encoding the text file based on that tree.

Contains 4 Classes:

Node: A class containing information about Nodes in the tree for encoding

SymbolCounter: A class containing a method to take a file and return a hash map of symbols and frequencies

HuffmanCoding: A class containing methods to construct a tree from smallest frequency to biggest, check a file exists, find a node based on its symbol, add symbols to a node if they don’t already exist encode and decode text.

HuffEncodingApp: A mini interface that also handles some of the file creation and makes the program easier to use.

Performance Analysis:

I used the books Frankenstein and Grim as my two choices. Frankenstein compressed to 55.3% (from 441KB to 244KB) of its original size and Grim also compressed to around 55.4% of its size (from 560KB to 310KB).

I struggled to use the tree of another file to encode a different file as it scrambled random characters in the text. My compression algorithm can’t use other trees to encode the text of a file. I did look at the compression rate of different languages and I found that in the book Grim it compressed to 55.4% in English, 58% (from 81KB to 47KB) in French and 58% in Portuguese (from 99KB to 57KB).

The data sets were significantly larger than the text files for the books, thus it took a lot longer to encode and decode the text files. Decoding, specifically was taking up to 20 mins for the largest of the data sets. For the Real dataset compression was at 67.6% of its original size (from 174.3MB to 258MB), the Pseudo-real data set was 65.6% of its original size (from 104.9MB to 68.8) and the Artificial data set was 12.5% of its original size (from 267.9MB to 33.5). These were great reductions despite the time taken.