ENGINEERING TRIPOS PART IIA

ELECTRICAL AND INFORMATION ENGINEERING TEACHING LABORATORY

MODULE EXPERIMENT 3F7

DATA COMPRESSION
Full Technical Report

Oliver Jones

Christ's College

Contents

1 Introduction

Data compression, or coding, is the act or reducing the storage size of digital data through algorithmically defined codewords. Such a process is crucial to the digital-age where a transmission channel needs to be utilised for as small amount of time as possible, or where vast amounts of information needs to be archived and stored without requiring significant hard drive or memory space on a digital device.

This Full Technical Report details how the algorithms laid out in the initial 3F7 lab can be modified to improve their performance. The main focus will be on making them adaptive, that is, how to create a compression algorithm that needs no prior knowledge of a probability distribution and calculates it on the fly. However some consideration will also be given on how to improve upon the independent and identically distributed (i.i.d) assumptions of the algorithms. As Shannon-Fano coding is largely of academic interest (as the optimality of Huffman makes it redundant), this report will not refer to modifications in the Shannon-Fano algorithm developed initially, and will instead focus on Huffman and Arithmetic coding.

2 Methodology

Code created in addition on top of the code in the lab is provided as an appendix, and entire source file can be found at `github.com/falcoso/CamZip`. An ipython notebook is also provided to demonstrate the code. The focus of the analysis of the algorithms in this lab will be the compression performance of the algorithms - complexity is taken into account comparatively with other algorithms, and the specific implementation of the algorithm provided in the Appendix may not be the most efficient implementation in runtime due to time of development constraints.

All algorithms developed were benchmarked against a number of files that can be found at `corpus.cantebury.ac.nz/`, summarised in Table 1.

For the adaptive algorithms presented in the benchmarking, N was taken to be 1% of the file length[1] and $\alpha = 0.5$. References to 'Entropy' mean the entropy of the file assuming an i.i.d. model and 'Markov Entropy' refers to the entropy of the file assuming a Markov chain model.

The Adaptive Huffman algorithm referenced in the Table 2 is the Vitter algorithm. The FGK algorithm provided does not decay its distributions so will be strictly worse performing (discussed further in Section 7). Discrepancies between functionality[2] of the algorithms was down to a focus on having an example of each type of functionality within a family of Compression algorithms, rather than each algorithm being equally capable.

[1] In reality such a varying of the N is not ideal as the information will also need to be passed to the decoder

[2] Estimators, Escape symbols, Decayed distributions etc.

3 Benchmarking Results

File	Size	Entropy	Markov Entropy	Desciption
a.txt	1	0	0	Single a characters
alice29.txt	148481	4.512876839	3.501779988	English text
asyoulik.txt	125179	4.80811622	3.417668127	As you like it script
bib	111261	5.2006763	3.364101213	Bibliography (refer format)
bible.txt	4047392	4.342751389	3.269098353	King James Bible
book1	768771	4.527148564	3.584519923	Fiction book
book2	610856	4.792632805	3.745217452	Non-fiction book (troff format)
E.coli	4638690	1.999821246	1.981416383	Genome of E. Coli bacterium
fields.c	11150	5.007698078	2.950248619	C source
grammar.lsp	3721	4.632267666	2.804585087	LSIP source
lcet10.txt	419235	4.622710675	3.559736297	Technical writing
news	377109	5.189631614	4.091895885	USENET batch file
paper1	53161	4.982982518	3.646101038	Technical paper
paper2	82199	4.601434708	3.522367548	Technical paper
paper3	46526	4.665104076	3.554867573	Technical paper
paper4	13286	4.699726052	3.477386059	Technical paper
paper5	11954	4.936154454	3.526041681	Technical paper
paper6	38105	5.009502936	3.611189809	Technical paper
plrabn12.txt	471162	4.477130818	3.442490717	Poetry
progc	39611	5.199016033	3.603359845	Source code in C
progl	71646	4.770085093	3.211590508	Source code in LISP
progp	49379	4.86877221	3.187521083	Source code in PASCAL
trans	91692	5.500996619	3.414075264	Transcript of terminal session
world192.txt	2408281	4.953090136	3.759446594	The CIA world fact book
xargs.1	4227	4.898431526	3.195171899	GNU manual page

Table 1: Properties of the Benchmark files

File	Stat. Huff	Ad. Huff	Stat. Arith	Ad. Arith	Cont. Arith	7zip
a.txt	0	7	2	9	3	952
aaa.txt	0	1.00006	0.00002	0.12741	0.00035	0.01704
alice29.txt	4.55529	4.57744	4.51289	4.54987	3.50206	2.61798
alphabet.txt	4.7692	4.81154	4.70045	4.81522	0.00039	0.01984
asyoulik.txt	4.84465	4.87094	4.80812	4.85613	3.41797	2.8514
bib	5.23171	5.27439	5.20069	5.26043	3.36444	2.20965
bible.txt	4.38495	4.37417	4.34275	4.33401	3.26911	1.7564
book1	4.56181	4.56293	4.52715	4.53015	3.58458	2.71718
book2	4.82339	4.78802	4.79264	4.75693	3.74529	2.2256
E.coli	2	2.24123	1.99982	2.0021	1.98143	2.04769
fields.c	5.0409	5.41865	5.0078	5.28951	2.95354	2.21776
grammar.lsp	4.66434	5.8358	4.63236	5.68584	2.81349	2.93684
lcet10.txt	4.65373	4.61886	4.62271	4.59406	3.55983	2.28365
news	5.22699	5.1962	5.18963	5.15768	4.09201	2.53349
paper1	5.01669	5.00581	4.98301	4.95265	3.64673	2.60928
paper2	4.6341	4.68534	4.60144	4.65941	3.52279	2.65959
paper3	4.68974	4.80705	4.66511	4.78743	3.5556	2.9427
paper4	4.73258	5.02047	4.69983	5.11712	3.4796	3.28948
paper5	4.97281	5.27137	4.93617	5.28836	3.52853	3.31738
paper6	5.04349	4.9556	5.00953	4.91613	3.61207	2.63902
plrabn12.txt	4.5196	4.52427	4.47713	4.49096	3.44257	2.80933
progc	5.23365	5.32577	5.19904	5.19767	3.60428	2.54818
progl	4.79936	4.72936	4.77009	4.71938	3.21208	1.68294
progp	4.89496	4.86004	4.86879	4.84953	3.18828	1.69141
random.txt	6	6.03048	5.9995	6.08046	5.97128	6.15168
trans	5.53526	5.40153	5.50101	5.3593	3.41452	1.47511
world192.txt	4.99637	4.9783	4.95309	4.93916	3.75947	1.6592
xargs.1	4.92382	6.07476	4.89875	5.87958	3.2018	3.55051

Table 2: Compression Rates for each algorithm on Benchmark files

4 Improving Arithmetic Codes

4.1 Adaptive Arithmetic Codes

Arithmetic coding works by splitting up smaller and smaller sub-intervals in the coding process based on the cumulative distribution of the alphabet [4]. There is no restriction for the cumulative distribution to be fixed between one interval division to the next. Therefore, an adaptive algorithm simply needs to take a frequency count of all symbols up to (but not including) the current symbol being encoded, and re-calculate the cumulative distribution every time. This is not a computationally intense process so a simple function was written to be shared between the encoder and decoder to ensure the cumulative distributions are calculated in the same way. The initial estimator used was a simple Laplacian estimator (see Section 6) with all 7 bit ASCII characters are initialised in the alphabet with an equal weighting.

4.2 Contextual Arithmetic Codes

The Static Huffman and Arithmetic codes presented in the original lab make the assumption that the data provided is i.i.d, this means that the maximum limit of compression is given

by the entropy of the source distribution $P_{X_n}(x)$ with alphabet $\mathcal{X}$:

$$H(X_n) = \sum_{\mathcal{X}} P_{X_n}(x) \log_2(P_{X_n}(x)) \tag{1}$$

However, it is clear that many files, such as English text, are not i.i.d but in fact will vary based on the text before it, for example $P(X_n = u|X_{n-1} = q) \approx 1$. Take a simple Markov chain as the assumed distribution of the data. The probability of a character only depends on the character before it (i.e. it has limited memory). From the definition of mutual information between the current symbol X_n and the previous symbol X_{n-1}:

$$0 \leq I(X_n; X_{n-1}) = H(X_n) - H(X_n|X_{n-1}) \tag{2}$$

$$H(X_n|X_{n-1}) \leq H(X_n) \tag{3}$$

This means that the compression limit for a model that takes probabilities based on the previous symbol encoded/ decoded will be strictly equal to or better than an i.i.d model.

Cormack et al [1] show that it is possible to apply Markov modelling to an arithmetic algorithm. The contextual algorithm provided creates a Markov chain model of the input data. The functions are provided with the transition matrix of a symbol going from one ASCII character to another, and after each symbol is encoded or decoded, the probability distribution is selected by indexing the row of the transition matrix corresponding to the current value, and filtering out any 0 probabilities. The results dramatically beat any of the other presented algorithms (by several orders of magnitude in the case of `alphabet.txt`) and in some files even beats the compression rate of the commercial compression algorithm '7zip'. However, these results do not factor in the need to provide the transition matrix to the decoder or the overheads of encoding the distribution information into the compressed file, so a practical algorithm will have slightly higher rates than this, but their impact will be small for large files.

Due to the nature of arithmetic coding, as file length increases, its compression rate become arbitrarily close to the entropy of the statistical model. As can be seen in Benchmarking Results, the compression rates agree closely with the calculated entropy of each file using a Markov chain model.

Theoretically an nth order Markov Process could be used to encode the data, however, an nth order model requires $|\mathcal{X}|^{1+n}$ worth of memory space to store the conditional data (where $|\mathcal{X}|$ is the alphabet size) so will quickly get out of hand and slow down the algorithm for more detailed models.

Witten and Bell [6] go into more detail on contextual methods including Prediction by Partial Matching (PPM) which is used to make the codes both adaptive and contextual but this has not been implemented directly here.

4.3 Terminating Arithmetic Codes

An ideal Arithmetic coding algorithm requires infinite precision as the interval of the code decreases as the length of the compressed file increases. The natural limitations of this on practical hardware are dealt with in the initial lab. However, this implementation of the algorithm can also add additional bits onto the compressed file due to the rounding inwards of the high and low boundaries of the interval being considered, reducing the dyadic interval by more than is necessary.

As a result, while the decoded file will contain all the information of the original file pre-compression, there will also be additional stray symbols decoded on the end that were not a part of the original file. The current implementation in the lab 'cheats' by supplying

the decoding function with the length of the original file so that it knows when to terminate, but there may be many applications when the real file length may not be known.

4.3.1 End of File Symbol

A messy but more straightforward approach to solving this problem is to create an additional 'End of File' (EOF) symbol that can be decoded and the decoding algorithm can be terminated on completion. This method has the main drawback of what probability should be assigned to this new symbol? Too small and you will add a significant number of bits to your code, too large, and the rest of your symbols will not be encoded effectively and your compression rate will drop dramatically. The provided adaptive Arithmetic algorithm does not make use of an EOF due to this trade off effecting the compression rate, particularly for smaller files. Instead Elias Gamma Coding is used.

4.3.2 Elias Gamma Coding

Elias Gamma Coding is a way of encoding integer numbers into binary that is completely prefix free. Devised in 1975 by Peter Elias [2] it is a very simple way of creating prefix free codes that can be appended to the encoded file and can be generated by the following recipe:

1. Let $N = \lfloor log_2(x) \rfloor$ where x is the integer to be encoded.
2. Append N zeros to the codeword.
3. Append the integer expressed in standard binary form to the codeword.

Decoding Elias Gamma coding is equally straight forward:

1. Count the number of zeroes before the first one as N
2. Read the next N bits including the first one
3. Read these digits as the standard binary representation of the integer that has been encoded.

This encoded integer can be added to either the start or the end of the file, so that when decoding occurs it can be decoded directly and the algorithm then knows how many symbols that need to be decoded. While this does add an overhead to any arithmetic coding file, this will disappear in the limit of large file sizes. The length of an Elias Gamma code is $2\log_2(N)$, so as N gets very large, the proportion of the overall code length will tend to 0. Moreover the overhead of an EOF termination will vary greatly depending on the initial weight that it is given as it will effect all other symbols to be encoded as well as itself by having an extra symbol in the file.

Both the Adaptive and Contextual Arithmetic algorithms provided make use of Elias Gamma Coding to know the source file length.

5 Adaptive Huffman Codes

Like Adaptive Arithmetic codes, an Adaptive Huffman code generates the probability distribution systematically as the data is encoded, so that when it is decoded, the probability distribution can be generated as it decodes, and no additional file is needed. Adaptive Huffman codes are commonly used for streaming data where there prior distribution is rarely known, and will often change over time. Streamed data also may not have a fixed end, so Arithmetic coding is not practical.

5.1 The Sibling Property

In an Adaptive algorithm, every leaf on the tree has a probability or weight assigned to it based on how often/ likely it is to appear in the message. Each internal node of the tree can also have a weight which is the sum of the weights of its children.

The sibling property states that each node (except the root) has a sibling and that each node can be listed in order of its probability with every node adjacent to its sibling. Gallager [3] demonstrates that any Huffman tree must obey this property, and any tree that obeys this property must be a Huffman tree. Adaptive Huffman algorithms can make use of this statement to modify trees by making the minimum changes to satisfy this property after an increment.

5.2 FGK Algorithm

Named after Faller, Gallager and Knuth - the inventers of this algorithm, the FGK algorithm demonstrates a simple method for generating the probability distributions, incrementing the counts of each symbol as they appear and then changing the structure of the tree based on these weights.

The algorithm described by Gallager can be summarised as follows [3] :

1. For a length K alphabet, create a list of $K-1$ sibling pairs. That is, create a list of adjacent nodes on a tree. Each sibling pair should hold 5 numbers.
    - 2 counters, 1 for each sibling node in the pair that is incremented when it is traversed
    - A Forward Pointer that points to the parent node of the sibling pair, with an additional bit to say whether it is a 0 or 1 traversal
    - 2 Backward Pointers that point to the nodes in the pair (either another sibling pair or a leaf node)
2. Create a list of the alphabet and the corresponding pointer that points to its location within the sibling list. This is effecitvely a list of leaf nodes.

For the algorithm to work as intended it is important that the list of sibling pairs is consistently maintained such that both the counts of the pair higher (or lower depending on the ordering) in the list than the current pair are both equal to or greater than the current counts. If not pointers are swapped around to make this the case.

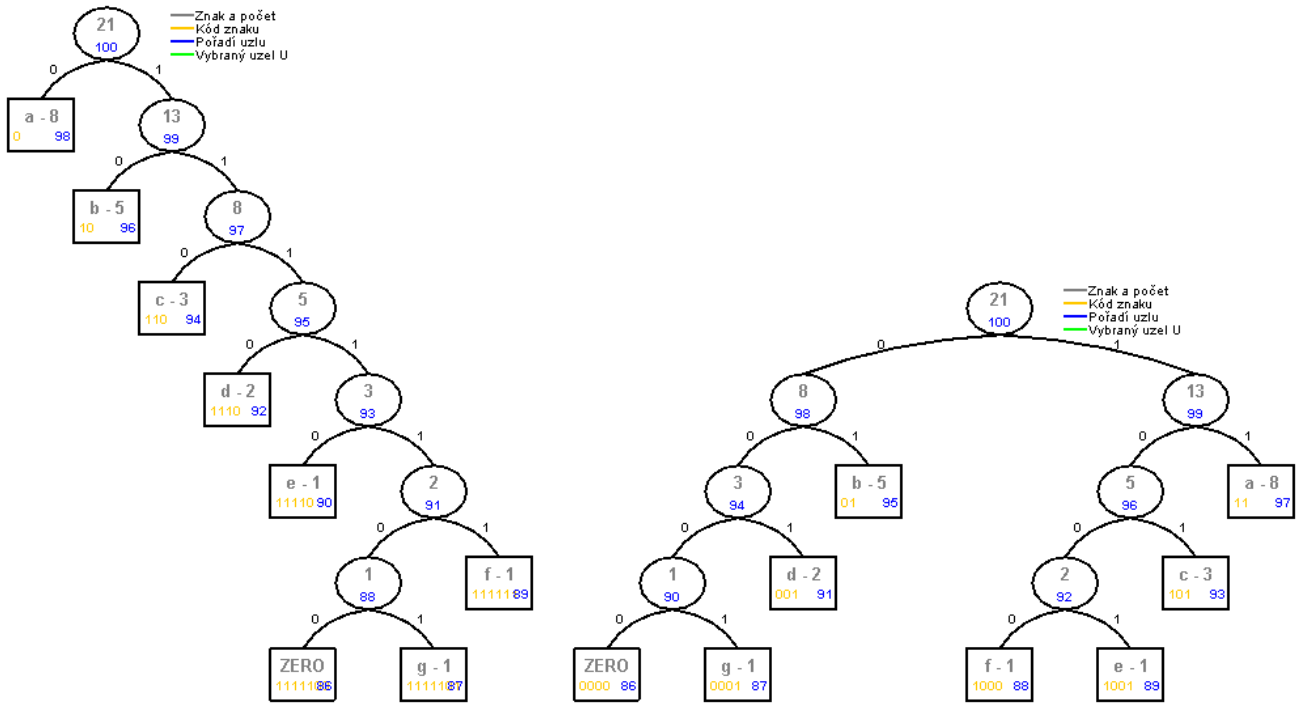
To make the code's implementation clear with reference to the above algorithm, a `SiblingPair()` class with attributes for each of the appropriate pointers is used. An additional bit was also added to the back pointers to say when the back pointer references the alphabet pointers list so the algorithm can switch between the two accordingly. This provided algorithm uses a Laplacian estimator (Section 6) and does not decay the distribution.

The algorithm is fairly straightforward to implement but does have the drawback that it will not always generate a tree of minimum weight, i.e. the depth of the tree will grow faster after a series of new symbols. An alternative, but more complex algorithm is the Vitter algorithm.

5.3 Vitter Algorithm

Building on the FGK algorithm, the Vitter algorithm takes the Adaptive Huffman algorithm one step further. Designed by Jeffrey Vitter in 1987 [5] it adds a few more properties to help 'balance' the tree and minimise its depth, as can be seen in Figure 1.

Each node in the tree has a given order, with the order of the root node being the highest and the order of the NULL node being the lowest (see Section 6). All nodes of the same weight are considered as being in the same 'weight class', with the order of all nodes in a weight class higher than the current weight class being greater than the order of all the nodes in the current weight class. When a weight is to be incremented, first the node is swapped with the node of highest order in the same weight class, then when the weight of the node is incremented it will then be at the bottom of the next higher weight class.



(a) FGK implementation (b) Vitter Implementation

Figure 1: Trees for the encoding of string 'abacabdabaceabacabdfg' with Zero node [3]

5.4 Complexity

Once a tree is generated in the static algorithm, a dictionary codebook is generated so a simple lookup is required to encode, and extended tree to decode. The Adaptive algorithm requires a tree for both cases, so that each node counter can be incremented as it is traversed so it can be re-created on decoding. Traversing the tree is much like a binary search algorithm similar to what is going on when a dictionary is indexed by it's key, but python is naturally optimised for searches within a dictionary compared to indexing multiple items in a list. In a lower level language (where compression algorithms are much more commonly implemented for speed), this will become less of an issue.

The real added complexity to the Adaptive algorithm is the shuffling of the trees after each symbol is encoded or decoded. Particularly at the start when the tree will be changing rapidly as each symbol is encountered for the first time, multiple nodes on the tree will be re-arranged as the counts change, slowing down the compression process. It is also noted that the change in N (see Section 7) is also going to have an affect on the runtime, as it determines

[3]Source: `http://www.stringology.org`, Blue, black and yellow numbers indicat the order, weight and codewords of the node respectively

how many times the sibling list is decayed over the encoding and decoding process - compared the runtime of the implementation when $N = 100$ vs. $N = 10000$.

5.5 Corruptibility

As demonstrated in the initial lab, the static Huffman algorithm is very robust to any corruption of the data. The change of a bit will change the current symbol and possibly one or two of the next symbols on encoding, but it quickly re-synchronises with the code and the rest of the encoding is completely un-affected. Adaptive Huffman however, is much more susceptible to corruption. As the tree is generated and the weights incremented the tree will change, so changing a bit in the compression code will change the weights on the tree in decoding from what it was at the same point in the encoding, and so the rest of the file becomes unreadable.

This will not always happen however - towards the end of the file when the estimated distribution approaches the real distribution (particularly for long files), the Huffman tree will not always change for each symbol encoded so may be able to cope provided the weight classes are significantly different from one another.

6 Biased Estimators and Escape Symbols

The main hurdle for Adaptive algorithms is the zero-frequency problem [6]. With an algorithm that generates the statistics of the message as it encodes, what does it do with a symbol it has not seen before?

6.1 Laplacian Estimator

As mentioned for the FGK and Adaptive Arithmetic algorithms, their implementations provided with this report initialise all symbols in the ASCII alphabet with equal weights - this is known as the Laplacian estimator. This has the advantage of requiring no block codes so will beat escape character solutions on shorter files, but the scope of the code is reduced to only the alphabet that it is initialised for as it has no way of dealing with any new symbols still. It also means that a file that does not make full use of the estimated alphabet will not be coded optimally.

6.2 Priori Estimator

If the compression is being optimised for human-readable text, an initial distribution can be assumed from an analysis of character frequencies across a given language, this can be a simple i.i.d distribution based on a simple frequency count of generic text sources or a more in depth contextual model. This has the advantage that the initial distribution is much more likely to be closer to the ideal distribution from the beginning, and so will not change dramatically over the course of an adaptive coding process. Nonetheless, it still has similar drawbacks as above and also not being optimal to non-natural language files.

6.3 Escape symbols

The Vitter algorithm provided makes use of an escape (or `NULL`) symbol that tells the encoder/ decoder that a new symbol has been encountered. The algorithm then breaks out of its normal algorithm and reads/ encodes the next set of bits as a normal block code (such as standard ASCII binary). The algorithm then adds this new symbol to its alphabet and continues as normal. Both the encoder and decoder will always have this escape symbol in its alphabet.

In the case of a Huffman algorithm, this will be at the lowest order on the tree, and when encountered will spawn two more nodes, with the 0 node being the new location of the escape symbol, and the 1 node being the new symbol that has been encountered.

As this method requires the transmission of the escape symbol *and* the block code of the new symbol, the compression ratio will naturally be worse for files with sizes comparable to the alphabet size contained with in it compared to initialising a redundant alphabet as described above. There could, of course, be a combination of the approaches applied, with an initial estimator and an escape symbol so that the scope of the algorithm isn't reduced.

7 Decaying the Distribution

To reduce the impact of the non-optimal initial estimator above, or adapt to a distribution change over the course of a file, the weights of each symbol can be decayed. As encoding continues, after every N symbols decoded, decay all the current values by multiplying by α where $\alpha < 1$. Any adaptive algorithm that doesn't decay its distributions will have a strictly worse compression rate, as the algorithm's distribution will tend towards the real distribution, whereas its static counterpart will be using the most optimal distribution for the assumed model from the beginning. Decaying the distributions like this effectively filters out the estimator so that the real distribution can take its place.

The choice of N and α determines how fast the distribution will change, the 'time constant' τ of the algorithm can be given by [3]:

$$\tau = \frac{N}{1-\alpha} \tag{4}$$

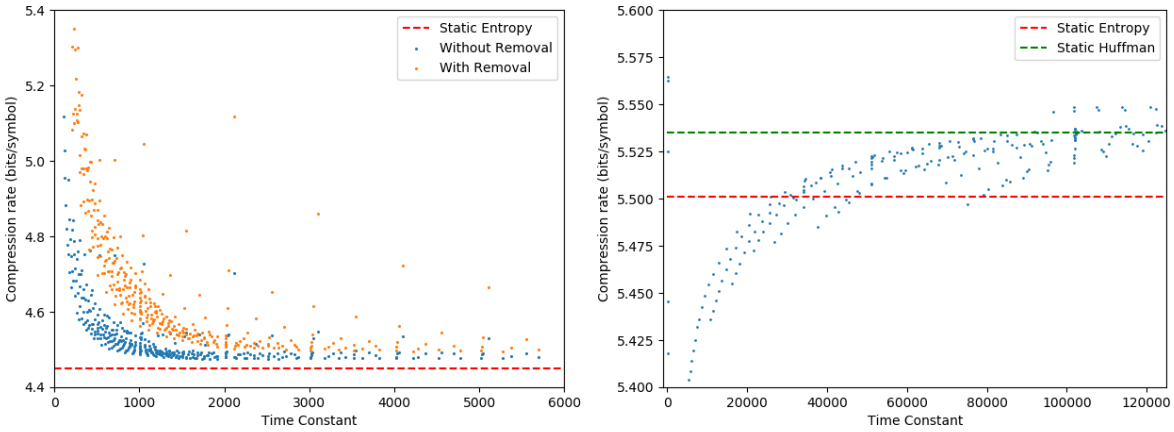
For Adaptive Huffman codes, even though α is not an integer, the counts themselves still need to be maintained as integers, such that when incrementing the weights of a node, it does not then skip several weight classes which may include the parent of the pair. This is naturally maintained by using a floor or ceiling function after all the counts have been multiplied by α.

The choice between floor or ceil will depend on whether you want to remove symbols that haven't occurred for a while and may be clogging up the tree. Ceiling will always mean the minimum value of a count is 1 once it appears on the tree, so there is no need to remove it. Floor will always reduce a count by at least 1, so that there is a point where a count can reduce to 0 and be merged with the `NULL` symbol. The provided Vitter implementation has the option for either with a boolean option `remove`.

Figure 2 shows the variation in compression rates as the time constant varies for different N and α for an Adaptive Huffman algorithm. While the adaptive compression of Hamlet cannot beat the assumed i.i.d source entropy, the compression of a terminal transcript can improve on this compression limit by a significant amount, implying its distribution changes over the file. Both sets of data appear to follow a trend that by empirical inspection is of the form:

$$C = |a\tau^{-1} - C_{stat} + C_{ad}| + C_{ad} \tag{5}$$

Where a is some constant, C_{stat} and C_{ad} are the max compression ratios of the Static and Adaptive algorithms respectively. In the case of Hamlet $C_{stat} \leq C_{ad}$ so the minimum for the algorithm tends towards Static Huffman as $\tau \to \infty$, whereas trans reaches a minimum at around $\tau \approx 3500$.



(a) Hamlet with and without node removal

(b) Cantebury 'trans' with node removal

Figure 2: Compression ratios of Adaptive Huffman for a given time constant as N and α are varied

8 Conclusions

Adaptive coding methods allow data to be decoded with no prior knowledge of the encoded data. This reduces the need to send additional uncompressed dictionaries with a coded file and with proper decay mechanisms, allow it to adapt to time-varying distributions.

While Arithmetic methods do beat Huffman methods as they become arbitrarily close to the file's distribution entropy, the nature of the algorithm means that it can only work on files of a fixed length, whereas Huffman codes can be used to compress data for streaming.

Adaptive methods often need an estimator of the initial distribution or an additional symbol to say when a new symbol is to be encountered. This does have the disadvantage of adding additional overheads to the file size, that a first pass algorithm would not, however the effect of these can be mitigated with decay mechanisms previously mentioned.

The main drawback of adaptive methods however is their susceptibility to corruption, while Arithmetic coding is already susceptible, Adaptive Huffman methods can also be corrupted if the encoder does not have the exact compression file.

8.1 Further Work

Algorithms presented in this report can be developed further to make them more standalone programmes. Further developments proposed are:

- Encode the information of N and α into the compressed file so that the information does not need to be shared between the decoder. A solution to this would be to use Elias Gamma coding to give N first, then a second Elias Gamma integer M for the precision of α, and then the next M digits being the bits of negative powers of 2.
- Combine Adaptive and Contextual algorithms into a single programme that calculates conditional probabilities on the fly. Further work can be used in PPM methods.
- Add escape symbols to the Adaptive Arithmetic algorithm to compare its results against a biased estimator.
- Implement Contextual models into the Huffman Algorithm

As alluded to in Section 7, there appears to be an empirical relationship between the N, α and the compression limit. Analysis of what is effectively an Infinite Impulse Filter in the decay mechanism and time varying statistics can be more closely looked into, to see if there is a method of accurately predicting the optimal time constant for a file.

References

[1] G. V. Cormack and R. N. S. Horspool. "Data Compression Using Dynamic Markov Modelling". In: *Comput. J.* 30.6 (Dec. 1987), pp. 541–550. ISSN: 0010-4620. DOI: 10.1093/comjnl/30.6.541. URL: http://dx.doi.org/10.1093/comjnl/30.6.541.

[2] P. Elias. "Universal codeword sets and representations of the integers". eng. In: *Information Theory, IEEE Transactions on* 21.2 (1975), pp. 194–203. ISSN: 0018-9448.

[3] R. Gallager. "Variations on a theme by Huffman". eng. In: *Information Theory, IEEE Transactions on* 24.6 (1978), pp. 668–674. ISSN: 0018-9448.

[4] J. J. Rissanen. "Generalized Kraft Inequality and Arithmetic Coding". In: *IBM Journal of Research and Development* 20.3 (May 1976), pp. 198–203. ISSN: 0018-8646. DOI: 10.1147/rd.203.0198.

[5] J. Vitter. "Design and analysis of dynamic Huffman codes". eng. In: *Journal of the ACM (JACM)* 34.4 (1987), pp. 825–845. ISSN: 1557-735X.

[6] I. Witten and T. Bell. "The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression". In: *IEEE Transactions on Information Theory* 37 (July 1991), pp. 1085–1094. DOI: 10.1109/18.87000.

Appendices

A Python Code

Attached with this report is a set of python files used in this report on top of those used in the initial lab. The start of each python file contains a docstring summarising the functionality of the algorithm (i.e. if it decays distributions, estimators used etc.). The manifest is as follows:

- Compression_demo.ipynb - A notebook demonstrating each of the algorithms discussed contained in the files below.
- adaptive_arithmetic.py - All the functions required for the Adaptive Arithmetic algorithm.
- context_arithmetic.py - All the functions required for the Contextual Arithmetic algorithm.
- fgk.py - All the functions required for the FGK algorithm.
- vitter.py - All the functions required for the Vitter algorithm.