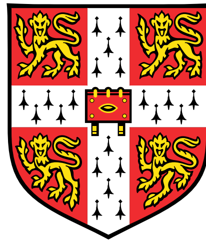
Stochastic Memory for Sequence Models

Making Good Compressors Use Less Memory



David Michael Goldfarb

Supervisor: Dr. Christian Steinruecken

Department of Engineering
University of Cambridge

This dissertation is submitted for the degree of
Master of Philosophy in Machine Learning and Machine Intelligence

Gonville and Caius College August 2022

To Eric and Gwen for their love and encouragement,
Adam and Rhianna for their hospitality and support,
and Gandalph "Randy" Goldfarb for his warmth and affection.

Declaration

I, David Michael Goldfarb of Gonville and Caius College, being a candidate for the degree of Master of Philosophy in Machine Learning and Machine Intelligence, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

This report makes use of the following software to aid in the production of results:

1. A reference implementation of Arithmetic Coding in Java.

   Availability: https://github.com/q4/ppm-dp/blob/master/java/Arith.java

   Purpose: This implementation was ported from Java to C++.

2. A reference implementation of CTW in Java.

   Availability: https://github.com/omerktz/VMMPredictor

   Purpose: This implementation was used as a means of providing black-box confirmation of correctness of a novel CTW implementation in C++.

3. A reference implementation of PPM-DP in Java.

   Availability: https://github.com/q4/ppm-dp

   Purpose: This implementation was used as a means of providing black-box confirmation of correctness of a novel PPM-DP implementation in C++.

Excluding the aforementioned software used in the production of results, all software used in this thesis was developed from scratch in both C++ and Python.

This thesis contains 14,761 words including footnotes, figure captions, and appendices: fewer than the 15,000 word limit prescribed by Degree Committee for the Faculty of Engineering.

David Michael Goldfarb
August 2022

Abstract

State of the art lossless compression techniques typically use memory intensive data structures which grow on the order of $\mathcal{O}(N)$ in the input size. Memory is an expensive resource, and many computing environments cannot afford $\mathcal{O}(N)$ memory scaling for typical workloads. There are several techniques that attempt to impose $\mathcal{O}(1)$ space restrictions on existing lossless compression techniques, which are similar in that they delete portions of their backing data structures. A novel approach presented in this thesis is to not delete the data structure, but instead allow *contamination* of statistics where each occurrence counter tracks an arbitrary number of entities.

The behaviour of a *contaminating* compressor is explored sparsely in contemporary literature. This thesis serves to document the characteristics of an implementation of a *contaminating* scheme when applied to several existing suffix tree-based compressors.

Summary of Contents

This thesis begins with chapter 1, an introduction of lossless compression suitable for a reader with a baseline knowledge of probability theory, with emphasis on the composition of coding and sequence modelling.

Then chapter 2 explores several existing compression methods: Context Tree Weighting (CTW), Deplump, and Prediction by Partial Matching with Dynamic Parameter Updates (PPM-DP) as well as their corresponding sequence models.

After which, chapter 3 presents an analysis of application of *contamination* on these existing methods with an emphasis on memory vs. compression effectiveness trade-offs.

Lastly, chapter 4 concludes this thesis with a discussion of areas of further investigation of *contaminating* compressors.

Contents

List of Figures 3

Nomenclature 4

1 Introduction 5

1.1 Lossless Compression . . . 5

1.2 Probability Foundations and Context . . . 7

1.2.1 How much can we compress? . . . 7

1.2.2 Entropy and surprise . . . 9

1.3 Coding . . . 11

1.3.1 Huffman Codes . . . 11

1.3.2 Arithmetic Coding . . . 12

1.4 Sequence Modelling and Compression . . . 17

1.4.1 Histograms: properties and design . . . 18

2 Existing Sequence Modelling Methods 23

2.1 Suffix Trees . . . 23

2.1.1 A formal description and demonstration . . . 23

2.1.2 Compacted suffix trees . . . 25

2.2 Context Tree Weighting . . . 26

2.2.1 CTW's sequence model . . . 26

2.2.2 Inference and learning in CTW . . . 27

2.3 Deplump . . . 28

2.3.1 SM: Deplump's sequence model . . . 28

2.3.2 Inference and learning in Deplump . . . 32

2.4 PPM-DP . . . 32

2.4.1 Classical PPM approaches . . . 33

2.4.2 PPM-DP's sequence model . . . 33

2.4.3 Inference and learning in PPM-DP . . . 34

3 A *Contaminating* Compressor **35**
3.1 Hash Tables 35
3.2 Existing Work 36
3.2.1 Constant-Size compressors 36
3.2.2 Hashing compressors 37
3.3 Random Hashing 38
3.4 Comparison with Amnesia 41
3.5 Maintaining Pure Histograms 44
3.6 The Influence of Full Updates 46
3.7 Concrete Trade-offs in Compression Effectiveness 46
3.8 Implementation Notes 49
3.8.1 Implementation of Amnesia 49
3.8.2 Implementation notes for suffix tree compressors 50

4 Conclusion **53**
4.1 Future Work 54

A Computational Details **56**
A.1 Efficiently Rescaling Narrow-Width Integers 56
A.2 Incremental Computation in CTW 56
A.3 boost::hash_combine and the Choice of a Non-Cryptographic Hash Function 57

B Compression Results on Full Corpora **60**

Bibliography **66**

List of Figures

2.1 Dense suffix tree of a partial message 24
2.2 Incremental suffix tree construction of a partial message 25

3.1 *contaminating* compression effectiveness over selected Calgary files 39
3.2 *contaminating* compression effectiveness over selected Canterbury files. 42
3.3 Comparison of Amnesia and *contaminating* compression effectiveness over selected Calgary files 43
3.4 Compression effectiveness of pure, low-order context histograms variant over selected Calgary files 45
3.5 Compression effectiveness of various updating schemes over selected Calgary files 47
3.6 Relative Compression Proportion Against Memory Usage over Shakespeare's corpus 48
3.7 Absolute Compression Proportion Against Memory Usage over Shakespeare's corpus 49

A.1 Compression effectiveness of various hashing schemes over the Calgary corpus 59

B.1 *contaminating* compression effectiveness over the Calgary corpus 61
B.2 *contaminating* compression effectiveness over the Canterbury Corpus 62
B.3 Comparison of Amnesia and *contaminating* compression effectiveness over the Calgary Corpus 63
B.4 Compression effectiveness of pure, low-order context histograms variant over the Calgary corpus 64
B.5 Compression effectiveness of various updating schemes over the Calgary corpus 65

Nomenclature

$\mathcal{A}$	An alphabet, or a collection of symbols
$\widetilde{\mathrm{P}}(x)$	The Exclusive Cumulative Distribution: $\mathrm{P}(X < x)$. The probability a random variable taking any value strictly less than x
$\mathrm{P}(x)$	The probability mass of a discrete random variable taking value x
$\langle x_n \rangle_1^N$	A sequence of N symbols
$\{x_n\}_1^N$	A set of N items, in no order.
$\mathcal{C}_c[s]$	The number of times a symbol s occurs in context c
$\lvert\mathcal{C}_c\rvert$	The total number of symbols seen in context c
$\lVert\mathcal{C}_c\rVert$	The number of unique symbols observed in context c
$[a, b)$	The half-open interval from a (inclusive) to b (exclusive)
$\mathbf{1}[x]$	The indicator function: equals 1 if the expression x is true, and 0 otherwise

Chapter 1

Introduction

1.1 Lossless Compression

A tremendous amount of data is being created every day from a variety of sources including people, sensors, and processes. Efficient storage and transmission of such data is an increasingly important concern in managing such amounts of data. One way to reduce the storage and transmission costs from data is through the use of *compression*: creating a (hopefully)[1] smaller representation of the data which can be stored or transmitted more cheaply, and from which the original can be fully recovered.

A *compression process* consists of three entities: a *message* to be communicated, a *compressor* which transforms the message into some encoded representation, and a *decompressor* which recovers the original message from the encoded representation.

For some types of data, such as pictures and audio, it is popular to discard perceptually-insignificant data and accept an approximate recovery of the message at decompression. Such techniques are termed *lossy* compression methods. However with data such as text documents and executable programs, lossy compression is not typically used. For example, in a picture, discarding high-frequency spatial data of an out-of-focus sky in the background may be acceptable, while the removal of key words in a legal contract is likely problematic.

Only techniques concerning *lossless* compression will be addressed in this thesis.

[1] A real compressor cannot reduce the size of all possible data and must necessarily make some data larger due to the pigeonhole principle. Typically a compressor is created to reduce the size of a particular set or kind of data, and it is deemed acceptable for such a compressor to fail at making other data smaller. This phenomenon will be detailed further in subsection 1.2.2.

Thus, refining the aforementioned definition of compression, *lossless compression* is a family of techniques to create a smaller, *perfectly recoverable*, representation of some data. Lossless compression techniques can be thought of as methods that remove redundant structure in data. Most lossless compression techniques can be deconstructed into two components: a *probabilistic model* for the message data, and a *coder*, which creates a more compact representation of the message using the model. The better the model predicts the data, the smaller the coder can make the compressed output. A clear advantage of delineating between a compressor's model and coder is the ability to manage the complexity of both components more effectively.

Other applications of compression

To mention other topics of interest in lossless compression which will not be investigated further in this thesis: Although the most common use of lossless compression is to create small representations of the same data, there may be further optimisation criteria placed on the encoded message. For example, in a Morse Code-like transmission scheme, there is a temporal component of data which consists of short *dits* and long *dahs*. Thus, it might be desirable for the compressed form to consist of more *dits* than *dahs*. Though this might seem contrived in the context of modern digital communication schemes, there are similar emerging design concerns investigations of low-power systems. For example, in the context certain memory systems which use the Bus-Invert method (Stan and Burleson 1995) more efficiently send bit streams with fewer transitions (a transition can be found in the bit strings "01" and "10" but cannot be found in the bit strings "00" or "11"). Thus, an effective compression scheme in such a domain might want to ensure a minimal number of bit transitions. Such additional optimisations will be considered as the domain of *channel coding* and be neglected for the rest of this thesis. For further discussion of a brief overview of early channel coding theory, consult Hancock and Holsinger (1962, Chapter 3).

For other applications of compressors beyond simply reducing the size of data, one such use is document classification (Marton, Wu, and Hellerstein 2005). As will later be discussed, a good compressor implicitly models data well. If there are several compressors which have been trained or designed to effectively compress different kinds of text, then an ensemble of such compressors may effectively classify a candidate document by identifying the constituent compressor that produces the smallest output.

1.2 Probability Foundations and Context

Let us begin with some essential definitions.

Let *data* be a collection of *messages*. A *message* m is a sequence of N *symbols* $x_1, \ldots, x_N$ from some source alphabet $\mathcal{A}_s$ and will be denoted as $m = \langle x_n \rangle_{n=1}^{N}$.

A *compressor* utilises a *model* and *coder*[2] to transform a message with symbols from $\mathcal{A}_s$ to message with symbols in some target alphabet $\mathcal{A}_t$.

A *model* (or *sequence model*, used interchangeably in this thesis) is, implicitly or otherwise, a probability distribution over symbols given a sequence of previously seen symbols, of the form: $\mathrm{P}\big(x_n \mid \langle x \rangle_1^{n-1}\big)$.

A *coder* is a means of transforming a sequence of source symbols into a sequence of target symbols given a model. There exist a variety of coders with different properties and capabilities, which will be discussed in section 1.3.

1.2.1 How much can we compress?

The structure and properties of data, which is captured through a model, dictate how much a message can be compressed.

For example, suppose we are interested in compressing messages that consist of a single symbol. Furthermore, assume that our source alphabet is uni-byte alphabet with($|\mathcal{A}_s| = 256$) and our target alphabet is binary ($|\mathcal{A}_t| = 2$). If there were no *a priori* knowledge of which symbols were more likely to appear, one optimal coding scheme would be the same "usual" binary encoding of the byte character. In this instance a compressed message would be no shorter than the source message.

Given that digital data is often encoded in a byte representation and compressors typically output binary at a bit-level granularity, *output bits per input Byte* ($\frac{b}{B}$) is the usual method for evaluating compression effectiveness. For this metric, much like the objective in compression, "lower is better".

Thus addressing the case above of transmitting a single, uniformly sampled character, the resulting compression effectiveness is $8\frac{b}{B}$.

Let us, however, assume that we have prior knowledge that only one of two symbols, a (`0x61`) or b (`0x62`), to use standard ASCII encoding, is to be compressed. If we are to assume that there is no bias towards one observing one source symbol over another, one optimal coding scheme would

[2] Occasionally, some compressors do not make an explicit delineation between their model and coder and instead propose an algorithmic process. For an intuitive explanation on how one can interpret the "model" of compressors which do not explicitly define one, see McFadden (1992, Lesson 1)

be a binary encoding, for example assigning a binary 0_2 for a and a binary 1_2 for b. In this instance, we can observe a compression effectiveness of $1\frac{b}{B}$.

Developing this two symbol case further: if messages containing a were known to be twice as frequent as messages containing b, can we do better than $1\frac{b}{B}$? This question will be answered in section 1.3 in the context of managing fractional bits of information.

If we were to alter this scenario and assuming that both we now used four input symbols (a, b, c, and d which were believed to be independent and have relative frequencies of $\frac{2}{3}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}$) and a message consisted of an *arbitrary* number of symbols, what would the compression situation be? Firstly, it is worthwhile to investigate how we would assign target encodings for each input symbol. Notably, if we were to take an arbitrary mapping, such as assigning the most frequent input symbol a one of the shortest encodings 0_2, and then arbitrarily assign some remaining encodings to b (1_2), c (00_2), and d (01_2), we would have difficulty decompressing most nontrivial target encodings. For example, under the aforementioned scheme, how should we interpret the encoding 001_2? Such an encoding could be reasonably decompressed into aab, ad, or cb: our encodings are not uniquely decodeable. An elegant way to ensure unique decodability (without expanding $\mathcal{A}_t$ with delimiter characters), is to ensure that any encoding scheme obeys the *prefix code property*, no encoding can be a prefix of any other encoding.[3]

Revising our scheme, we could still assign a a short encoding 1_2, but assign $b \to 01_2$, $c \to 000_2$, and $d \to 001_2$, arbitrarily. Following through, how can we interpret this scheme's compression effectiveness? One such method is to compute the expected target message length per input symbol, in other words:

$$\begin{aligned}
\mathbf{E}\left[\frac{b}{B}\right] &= \frac{1}{8} \cdot \sum_{x \in \mathcal{A}_s} \mathit{NumOfTargetBits}(x) \cdot \mathit{freq}(x) \\
&= \frac{1}{8}\left(1 \cdot \frac{2}{3} + 2 \cdot \frac{1}{3} + 3 \cdot \frac{1}{3} + 3 \cdot \frac{1}{3}\right) \\
&= \frac{1}{8} \cdot \frac{10}{3} \\
&= \frac{5}{12}
\end{aligned}$$

Notably, if we had no prior knowledge about the relative frequencies of each symbol, but were to make the same symbol assignments, we would see

[3] Prefix codes constructed from symbol frequencies are a well known method and have been explored by Fano (1949), Huffman (1952), and Shannon (1948).

an expected $\frac{b}{B}$ of

$$\mathbf{E}\left[\frac{b}{B}\right] = \frac{1}{8} \cdot \frac{1}{4}(1+2+3+3) = \frac{36}{8} = \frac{9}{2} = \frac{54}{12}$$

Thus, we see that effective symbol assignments that are based on symbol frequencies can make a significant difference in compression effectiveness. Looking at the case of four symbols with skewed frequencies, we can still expect an output of slightly less than half $\left(\frac{5}{12}\right)$ a bit for every incoming byte.

In order to understand how this pattern generalises for other data with potentially varying knowledge of relative symbol frequencies, we must further explore our notions of a model.

1.2.2 Entropy and surprise

Given the above definitions of a message and a symbol, let us assume we also have a known symbol probability for each symbol in the message $\mathrm{P}(x_n)$ according to some model. When symbol occurrences are independent, the probability of a message m can be factorised as follows:

$$\mathrm{P}\left(\langle x_n \rangle_1^N\right) = \prod_{n=1}^{N} \mathrm{P}(x_n)$$

Shannon (1948) notes from Hartley that taking the logarithm of an event's reciprocal probability has convenient properties, namely the use of a logarithm with a base of two causes values of such an expression to take the units of binary digits or *bits*.

The quantity $\log_2 \frac{1}{\mathrm{P}(x_n)} = -\log_2 \mathrm{P}(x_n)$ is termed the *information content* associated with a symbol x_n under the model P. If a symbol has a high information content, the model views it as rare and surprising, and if a symbol has a low information content it is deemed common and unsurprising.

For example, seeing the number 1 on a fair, four-sided die is an event with a probability of $\frac{1}{4}$ and an information content of $\log_2 \frac{1}{\frac{1}{4}} = \log_2 4 = 2$ bits. It takes two binary digits to encode any one of the states of a fair, four-sided die.

The probability of seeing an a from the biased set shown in subsection 1.2.1 was $\frac{2}{3}$. The associated information content is $\log_2 \frac{1}{\frac{2}{3}} = \log_2 \frac{3}{2} \approx 0.585$ bits. Similarly, seeing any single rarer symbol: $\{b, c, d\}$ has a corresponding information content of $\log_2 \frac{1}{\frac{1}{8}} = \log_2 8 = 4$ bits.

The information content roughly corresponds to the length of the prefix codes assigned to each symbol.

To extend this result to a message, the information content metric has the useful characteristic of being additive due to the product-to-sum property of logarithms. For example, a message consisting of five independent occurrences of a ($aaaaa$) has an information content of: $5 \cdot \log_2 \frac{1}{\frac{2}{3}} = 5 \cdot \log_2 \frac{3}{2} \approx$ 2.92 bits. However, if we were to attempt to form an encoding of that message using the scheme described at the end of subsection 1.2.1, where each symbol was allocated a fixed-size code word, we would use five bits (11111_2).

Note that because this prefix coding scheme uses an integer number of bits for each symbol, coding long messages with many symbols that have fractional information content quickly becomes expensive. Additionally, this scheme is non-adaptive: we currently have no effective means to adjust our prefix code if we know that $\mathrm{P}\big(x_n \bigm| \langle x_i \rangle_1^{n-1}\big) \neq \mathrm{P}(x_n)$. Efficient design of coding techniques which can both efficiently handle fractional information content and adaptive symbol probabilities will be discussed in section 1.3.

In general, when given a probability distribution over messages, we can compress a message to a sequence of bits whose length is the message's information content (rounded up to the nearest integer).[4]

Information content is always *relative to some model* because the model defines $\mathrm{P}(x_n)$ and therefore the information content of a message. Thus, our model determines how much a given message can be compressed: the better the model matches the data, the better the data can be compressed.

Adversarial inputs and poor models

As a brief tangent, if a model consistently assigns high probability to symbols in a sequence, it means the information content of the message's constituent symbols is low, thus implying the message is "unsurprising" to the model. Moreover, given a model, it is possible to analyse what messages would be surprising. For example, given the model described at the end of subsection 1.2.1, a message consisting entirely of the symbols $\{b, c, d\}$ would be very surprising, and have a high information content. Specifically, the message "*dddddddddd*" of length 10 would have an information content of

$$10 \cdot \log_2 \frac{1}{\frac{1}{9}} = 10 \cdot \log_2 9 \approx 31.7 \text{ bits}$$

Under a uniform model, this same message would have an information content of

$$10 \cdot \log_2 \frac{1}{\frac{1}{4}} = 10 \cdot \log_2 4 = 20 \text{ bits}$$

[4] Though, usually slightly longer due to inefficiencies in the chosen coding scheme.

If one were using the encoding of this alphabet induced by a uniform model, the compressed message would be *smaller* than the scheme described above.

Certain adversarial, or surprising inputs can result in a compressor producing *longer* "compressed" messages relative to the input message length. This adverse outcome highlights the importance of constructing appropriate models that accurately reflect the distribution of messages to be compressed.

1.3 Coding

As we have seen from subsection 1.2.1 and subsection 1.2.2, two properties we would like in codes are the ability to communicate efficiently symbols that have a fractional amount of information content and to utilise effectively adaptive symbol probabilities.

This section will first give a more general construction for prefix codes, in order to provide a foundation from which to further analyse various coding schemes. This section will then discuss arithmetic coding, which satisfies both of the aforementioned properties.

1.3.1 Huffman Codes

As described in subsection 1.2.1, a reasonable means of assigning codes to symbols with known probabilities is to give the most common symbol the shortest possible encoding, then give the next most common symbol the next shortest possible encoding while obeying the prefix property.

Huffman (1952) describes an one algorithm that uses this intuition and takes the approach of working backwards from the most rare symbols.

Firstly, the least common two symbols are selected and "joined" to a node. This node is considered as a new, virtual symbol whose probability is the sum of its two children. The next two least common symbols (including virtual symbols) are then joined, and this step is repeated until a single node remains. The resulting structure is a binary tree, often called a "Huffman Tree". To encode a symbol using this tree, treat all left child and right child traversals as 0 or 1 respectively, and each symbol's encoded value corresponds to the path needed to reach the corresponding symbol through a series of left and right child traversals.

The use of single-symbol prefix codes is problematic when attempting to encode messages of *more than* one symbol due to the compounding "overspending" of whole bits on symbols with fractional bits of information content. Notably, however, there will be no "overspending" in the special case where all symbol probabilities are reciprocals of powers of two which owes to

the paired nature of node construction in this scheme. Under such circumstances, Huffman Coding is optimal.

1.3.2 Arithmetic Coding

Shakespeare on a ruler

"All of the works of Shakespeare can be encoded as a single point on a ruler". This phrase provides a reasonable spatial intuition for the operation of an arithmetic coder and the following example describes the operation of such a coder.

Assume that $|\mathcal{A}_s| = 30$ (26 characters of the alphabet, three punctuation characters, and a space). Take a 30 cm ruler, and partition it into 30 uniformly-sized units, with the first corresponding to a, the interval of $[0, 1)$, the second corresponding to b $[1, 2)$, and so on, with the last corresponding to a space $[29, 30)$. Given this partitioning, select the unit that corresponds to the first letter of the first work of Shakespeare. Assuming we've selected Shakespeare's first sonnet, this letter is "f" (of the word "from") that corresponds to the unit $[5, 6)$. Now, divide the interval $[5, 6)$ further into 30 uniformly-sized units (each will be of size $\frac{1}{30}$), and select the subdivided unit corresponding to the next letter. The letter "r" corresponds to the unit $\left[5\frac{17}{30}, 5\frac{18}{30}\right)$ $\left(\left[5.5\overline{6}, 5.6\right)\right)$. This process repeats until all letters have been processed, culminating in a narrow interval.

Any sequence of characters can be encoded to a designated region of the ruler. Choose a point inside this region (not on the boundary) that has the shortest decimal expansion.

A recipient could recover the original message from the decimal number by following the same process forward, but instead of selecting the interval that corresponds to the letter, the recipient selects the letter corresponding to the interval.

To illustrate, suppose the recipient received the target number 5.567. After constructing the initial set of 30 intervals, a recipient would see that the target number resides in the interval $[5, 6)$, and record that the first letter was an "f". The recipient would then construct a set of $\frac{1}{30}$ cm large intervals from $[5, 6)$, see that the target number resides in the interval $\left[5.5\overline{6}, 5.6\right)$, and record that the second letter was an "r". The recipient would then note that the number terminates, and stop decoding. Thus, through the above process, the target number can be decoded into the original text.

An infinite precision Arithmetic Coder

Arithmetic coding, described in Pasco (1976), Rissanen and Langdon (1979), and Rissanen and Langdon (1981), and a closely related variant called Range coding, introduced in Martin (1979), rely on the same fundamental operations, but uses an original range of $[0, 1)$. Furthermore, the lengths of subintervals at each step are not required to be uniform, and the number of intervals at each step is allowed to vary. The former confers the ability for arithmetic coding to manage adaptive symbol probabilities, and the latter property gives arithmetic coding the ability to manage dynamic alphabets.[5]

Adaptive symbol probabilities are handled formally by requiring a Cumulative Distribution Function (CDF), and an associated symbol ordering,[6] to be placed over the alphabet at a particular index in the message. In particular, an *Exclusive Cumulative Distribution* $\mathrm{P}(X < x) = \widetilde{\mathrm{P}}(x)$ (rather than $\mathrm{P}(X \leq x)$), is valuable as its image corresponds exactly to the half-open semantics used in the above "Shakespeare on a Ruler" example. In particular, the $\widetilde{\mathrm{P}}(x_n)$ corresponds to the inclusive lower bound of a symbol's interval induced by a model. In order to acquire the half open range, one can use the probability mass of the symbol $\mathrm{P}(x_n)$, to form $\left[\widetilde{\mathrm{P}}(x_n), \widetilde{\mathrm{P}}(x_n) + \mathrm{P}(x_n)\right)$

An additional wrinkle that must be considered is the stopping criteria or terminating conditions, in other words, communicating the length of the message.

Modern coders have developed two possible responses to communicate the length of the given encoded message. The first is directly communicating the (possibly encoded) length at the start of the message and the second is encoding an out-of-alphabet, end-of-file (EOF) symbol at the conclusion of a given message. A decoder using the first method would stop decoding after producing the number of tokens specified by the initial message length,

[5] A system that manages a message with a dynamic $\mathcal{A}_s$ be implemented in terms of a system that handles a single, large, statically-sized $\mathcal{A}_s$, but assigns certain symbols a probability mass of zero at different indices in a message. Thus in a sense, the second property is a consequence of the first property. Dynamic alphabets will not be of significant discussion for the remainder of this thesis.

[6] Symbol ordering presents an interesting engineering design decision when implementing any coding scheme that requires continuous construction or maintenance of symbol probabilities. For decoder implementations that perform a linear search through a given alphabet's associated CDF (in order to assign a symbol given a point in some range), it is often advantageous to order symbols in terms of decreasing probability mass (or increasing if searching is done backwards). Such an ordering can potentially speed up searches for common symbols (according to the given model), as the symbol with the greatest probability is found first. However, for the remainder of this thesis, the usual ordering imposed by typical definitions of the specific $\mathcal{A}_s$ (typically lexicographically) will be used unless otherwise stated.

while a decoder using the second method would stop decoding right before attempting to emit the EOF symbol. A third, but non-universal, method to communicate stopping criteria is to rely on the semantics of the message itself. One example is that many digital file formats include a header which contains file-length, which could theoretically be incorporated into the decoding procedure. Additionally, many text files are terminated with an in-alphabet *NULL* terminator, which can be used instead of an out-of-alphabet *end-of-file* marker. Both solutions are amenable to ergonomic implementations, but the main difference between these two termination methods lies in the implicit difference in modelling of message length. Specifically, the EOF solution implies that the same model which analyses the message itself is suitable for analysing message length. In contrast, the explicit message length approach enables explicitly modelling the message length separately from the model used for the message content. Both approaches have reasonable theoretical and practical justifications and both methods are used frequently in practical coders.

When analysing the "Shakespeare on a Ruler" example under this more rigorous light, one may observe that the nth accumulated lower bound of the interval, L_n takes the form of:

$$L_n = \prod_{i=0}^{n} \widetilde{\mathrm{P}}(x_i)$$

and the nth accumulated upper bound of the interval, U_n takes the recursive form of:

$$U_n = U_{n-1} \cdot \widetilde{\mathrm{P}}(x_n) + \mathrm{P}(x_n)$$

which was an observation made jointly by both Rissanen (1976) and Pasco (1976).

The above description describes an infinite precision arithmetic coder, which can be naively implemented in any programming language that supports infinite precision arithmetic ("bignums"). Analysing the computational complexity of such a procedure, it appears to be $\mathcal{O}(N)$ in the length of the message if $\widetilde{\mathrm{P}}(x)$ and $\mathrm{P}(x)$ can be calculated in $\mathcal{O}(1)$. However, arithmetic for interval narrowing is not $\mathcal{O}(1)$ in number of digits in $\{L_n, U_n\}$ for standard bignum implementations. Instead arithmetic and is typically $\mathcal{O}(N)$ in the number of digits in each operand. Thus, given that the number of digits in $\{L_n, U_n\}$ grows linearly in the number of symbols seen thus far, an infinite precision arithmetic coder has a complexity of $\mathcal{O}(N^2)$ in the message length. For even modestly-sized files on the order of megabytes, quadratic complexity is highly impractical.

However, the complexity of arithmetic coding need not be quadratic if arithmetic can be performed in constant time. In fact, a finite-precision arithmetic coder, which utilises constant-time arithmetic and finite-precision representations of $\{L_n, U_n\}$ can be constructed with a few modifications.

A finite-precision Arithmetic Coder

The key modification to make to an infinite-precision arithmetic coder is to conduct all arithmetic in a *scaled, integral* domain such that the interval of reals in $[0, 1)$ maps to the interval of integers in $[0, 2^p)$ where p is typically a machine word size (*i.e.* 32 or 64 for most modern, non-embedded hardware).

Scaling to integer arithmetic does have the immediate restriction that no symbol probabilities smaller than $\frac{1}{2^p}$ (the minimum "resolution" of the integral domain) can be represented, but this is typically well-tolerated by most compressors.[7] Given this finite scheme, rescaling must also be employed to ensure that the minimum resolution of an interval does not grow too large. In fact, rescaling is also coupled with "flushing" or outputting bits to a buffer, which constitutes the compressed representation of the input message.

Elaborating on the rescaling and flushing procedures further, consider the midpoint of $[0, 2^p)$: $\frac{2^p}{2} = 2^{p-1} = M$. If a number in the interval $[L_n, U_n)$ is completely contained in $[0, M)$, the fractional binary expansion of a number in that interval starts with a zero (as it must be strictly less than $0.1_2 = \frac{1}{2}$). Similarly any interval completely contained in the range $[M, 2^p)$ implies that the fractional binary expansion of a number in that interval must start with a one (as it must be at least as big as 0.1_2). If $U_n < M$ a zero is flushed, and if $L_n \geq M$ a one is flushed. In both cases, the interval is rescaled either by halving, or re-centering via subtraction by M followed by halving.[8] Collectively, these two cases can be termed the "standard rescaling" cases.

Such a procedure is almost complete, but neglects the case where $\{L_n, U_n\}$ increasingly approach M (and thus each other), but never meet one of the two above conditions. In pathological cases where $\{L_n, U_N\} = \{M - 1, M\}$, there is likely not enough mass to represent $\widetilde{\mathrm{P}}(x)$ faithfully.[9] To counteract

[7] This tolerance often owes to the fact that many compression systems give all symbols a minimum bin-size typically much greater than $\frac{1}{2^p}$.

[8] Because $\{L_n, U_n\}$ are in the integral domain, halving can be efficiently conducted via left-bit-shift-by-one.

[9] Such a harmful case can be reached by relatively innocuous data which has a run of symbols whose resulting intervals contain M. Although the pathological case cannot be reached with a uniform model over $|\mathcal{A}_s| = 30$, it could be reached in the case of $|\mathcal{A}_s| = 29$ (no space) and repeated selection of the character o (corresponding to the range $[14, 15)$, which contains $M = 14.5$). Such a sequence may be unusual for human text, but would be unsurprising for machine-data.

such pathologies, it is necessary to re-scale in this "centre-approaching" case as well, which is slightly more involved than the "standard rescaling" cases.

Let the first/lower quarter be defined as $LQ = \frac{1}{4} \cdot 2^p$, and the third/upper quarter be defined as $UQ = \frac{3}{4} \cdot 2^p$. If $L_n > LQ$ or $U_n < UQ$, both $\{L_n, U_n\}$ are re-centred via subtraction of LQ and then divided by two. This operation corresponds to "zooming-in" to the centre of the interval. However, instead of flushing, a pending/waiting counter is incremented. In either of the above "standard rescaling" cases, the flushing operation corresponds to emitting the specified bit, followed by the complement of that bit repeated N times, where N is the pending/waiting counter, and concluded by clearing the pending/waiting counter. A rough intuition of this bit flipping behaviour can be gleaned from the fact that if outputting a single 0 or 1 happens when L_n is either 0.0_2 or 0.1_2 (corresponding to the intervals $[0, 0.1_2)$ and $[0.1_2, 1.0)$ respectively), then L_n when picking points defined by quarter points are 0.01_2 or 0.10_2 (corresponding to the intervals $[0.01_2, 0.10_2)$ and $[0.10_2, 0.11_2)$ respectively), or when "zooming-in" further, the points 0.011_2 or 0.100_2 (corresponding to the intervals $[0.011_2, 0.100_2)$ and $[0.100_2, 0.111_2)$ respectively). Note how in all of the above cases, L_n can be described as a 1 or 0, followed by some number of repetitions that digit's complement.

A passing note from this analysis is that given the three conditions for rescaling, $\{L_n, U_n\}$ can only become as close as $M = UQ - LQ$. This minimum difference demonstrates that the aforementioned minimum permitted size of symbol probabilities of $\frac{1}{2^p}$ is an overestimate. When considering edge cases, finite-precision arithmetic coding can only correctly handle probability masses which are at least as big as $\frac{1}{2^{p-1}}$.[10]

The aforementioned procedure describes a finite-precision arithmetic coder whose complexity is $\mathcal{O}(N)$ in the length of the message to be compressed.

Comparing Huffman and Arithmetic Coding

Because Huffman Coding produces potentially sub-optimal compressed messages because each source symbol has a one-to-one correspondence with each (bit-aligned) target codeword. This property, which is powerful for spatially efficient random access,[11] can yield very poor compression if the information

[10] See footnote 7 which explains why this is likely a non-issue.

[11] Although Huffman Codes are of variable length, random access can be enabled by simply knowing the particular offset at key checkpoints in the message (say the index/offset of every 1,000 characters) and decoding from there. Thus, random access can be naively enabled via auxiliary information, though more sophisticated methods obviate the need for such bookkeeping, see Yang, Lin, and Hu (2018). One may think that the serial and recursive structure of arithmetic coding inhibits any attempt at random access, however auxiliary information here too can enable random access: it simply requires a message

content of a common symbol is slightly more than a whole number of bits, as the number of excess bits sent will grow linearly with the occurrence count of that symbol. In arithmetic coding, where a single target symbol can represent multiple source symbols, the resulting compression is nearly optimal, being typically no more than two bits greater than the message's information content.[12]

Thus, given an effective means to reduce a message to nearly the size of its information content, the key path to good compression is good sequence modelling.

1.4 Sequence Modelling and Compression

As stated earlier, a sequence model is a probability distribution over the symbols in an alphabet given previously seen symbols:

$$\mathrm{P}\left(x_n \mid \langle x \rangle_1^{n-1}\right)$$

Such a probability distribution can be used for prediction, or for estimating future events given past data. As seen in section 1.3, if $\mathrm{P}(x_n)$ is consistently close to 1, then the message's information content is small, and a coder is able to encode such symbols in very few bits.

An "improvement" one may consider in designing a good sequence model is to use the entire message m (rather than only the symbols $\langle x_i \rangle_1^{n-1}$) which may be more effectively modelled. Practically speaking, additional content on which to condition may indeed produce a better probability model, however the compression process would be violated by such a construction. The decompressor only has access to symbols which have already been decoded, and thus would not be able to replicate the same $\mathrm{P}\left(x_n \middle| \langle x_i \rangle_1^N\right)$ given only $\langle x_i \rangle_1^{n-1}$.[13]

offset as well as all internal state ($\{L_n, U_n\}$) of the coder.

[12] As described in Witten, Neal, and Cleary (1987), excess bits in arithmetic coding come from a combination of extra bits needed to encode termination, rescaling, and the use of finite-precision arithmetic. For reasonably-sized messages, this overhead is negligible and well-tolerated.

[13] Note that a compressor could technically skirt this wrinkle by encoding additional information about future symbols in a *header* which encodes future model information. Practically, "additional information" implies sending pre-trained model parameters along with the encoded representation of the message, which must also contribute to the total length of the encoded message. When accounting for this fact, it has been shown in Steinruecken (2014, Section 4.4) that encoded representations with a header are no more spatially efficient than plain encoded representations which are decoded with no prior header.

Another important consideration is that all sequence models are imperfect. Typically, real data such as human text is too complex to be modelled completely by most classical sequence models. As suggested in section 4, a mismatch between the data and the model can result in poor compression.

Lastly, it is important to note what general implementation family the below-mentioned models will follow. All models discussed in this thesis at a high level take the following approach. Messages are read and "learned" symbol-wise, and each symbol is used to update one or more *histograms*, the statistical bookkeeping structures of most models. Before updating histograms for the nth symbol x_n, a model is capable of generating a conditional probability $\mathrm{P}\big(x_n \mid \langle x_i \rangle_1^{n-1}\big)$ which can be used by a compressor to make coding decisions. The importance of generating and coding a given symbol with a probability density *before* that symbol is learned is to obey the compression process. The decompressor needs to replicate the procedure of learning and at any given time it only has access to the previously decoded symbols.

1.4.1 Histograms: properties and design

This section describes a histogam's underlying semantics, internals, efficient storage, efficient access, effective transformation into a conditional probability, and mechanisms of update.

Histogram semantics

In the context of compression, a histogram counts the occurrences of symbols in a specific context. In the case of all models described below, contexts are the (finite-length) sub-sequences immediately preceding the current symbol.

Histogram internals and storage

Internally, a histogram typically consists of at least a set of counters of symbol occurrences for its context. Sequence models commonly group histograms that share the same prefix, or initial $N-1$ gram. For example, given $\mathcal{A}_s = \{a, b, c\}$, a particular histogram may track the distinct counts for the prefix-sharing 3-grams: $\{aba, abb, abc\}$ which occur a given message.

Histograms may also contain additional information which can broadly be described as cached or incremental results. Such incremental results may include items such as the sum of all counts, the number of unique N-grams tracked, or the transient results of more involved operations on other, related histograms.

Because a given sequence model typically manages a huge number of histograms (the order of 10^6 is typical), efficient storage of these histograms is also a chief concern. To illustrate the immense size of such models, imagine a model that utilises a $|\mathcal{A}_s| = 256$ uni-byte alphabet. Assuming counts are tracked by integers with a width of W Bytes, the size of such a model is $256^{N+1} \cdot W$ bytes. Taking $W = 4$ bytes and the typical RAM of a modern computer to be roughly 32 GiB, N can be no larger than roughly 3. Thankfully, most real data does not induce a "full" or "dense" N-gram model, and the sparsity of real data can be exploited to practically construct N-gram models for $N \geq 3$. One such exploitation is to allocate histograms lazily.[14] Additionally, histograms may choose not to cache intermediate results, opting to do all calculations "from scratch".

Furthermore, counts can be represented with narrow-width integer types, such as $W = 1$ byte. As a side remark, it is not unreasonable to believe that such narrow representations of counts may be unsuitable for effective sequence modelling, because, especially for low-order histograms (say, of order-0), it is very likely to see more than $2^8 - 1$ occurrences of a symbol in a given message. There are three reasonable means of handling larger counts in narrow-width arithmetic: wraparound, saturation, and rescaling.

Wraparound ($255 + 1 = 0$) is the default semantics of unsigned integer overflow, but it presents poor compatibility for good sequence modelling, because common symbols get "reset" and are then considered as uncommon symbols. Saturation ($255 + 1 = 255$) is another natural choice for managing large values in unsigned arithmetic, which presents less poor compatibility, as in the limiting case prediction will degrade to a uniform distribution. Lastly, rescaling ($255 + 1 = 128$, all counts divided by two, rounded up)[15] presents the property that relative magnitudes between symbol frequencies are preserved, and lends hysteresis prevention attributes to a given model, as long runs of a single symbols can quickly be forgotten if needed. However, this hysteresis prevention property can be harmful too, because long runs of single symbols can cause a loss of model accuracy for other symbols.

Histogram access

Most sequence models that utilise histograms for managing symbol occurrence counts in N-gram contexts typically access histograms of increasing prefix-length. For example, a model operating over $\mathcal{A}_s = \{a, b, c\}$ which has

[14] Counts within histograms can be lazily allocated too.

[15] Initially this scheme may be thought to oppose efficient hardware implementation as generic integer division is typically a high-latency operation in common microarchitectures. However such scheme can indeed be efficiently implemented, see section A.1.

seen the sequence a, b, c, c, and is being queried for the probability of seeing an arbitrary character $*$, $\mathrm{P}(* \mid a, b, c, c)$, would typically consult the histograms for the N-gram contexts: $\{\{*\}, \{c, *\}, \{c, c, *\}, \{b, c, c, *\}, \{a, b, c, c, *\}\}$

The organisation of such histograms is implementation-dependent and should not affect correctness of the resulting probability, but typically two structures are employed.

The first is that of a hash table, where the key is some function of the context. Hash tables, particularly the design trade-offs they present in sequence models, will be discussed in section 3.1. The second is that of a suffix tree,[16] where each histogram connects to all histograms of larger prefixes by one symbol. For example, the histogram named $c, *$ has "child-pointers" to the set of histograms $\{(a, c, *), (b, c, *), (c, c, *)\}$. Suffix trees, and the design trade-offs they present in sequence models will be discussed in section 2.1.

A minor, but relevant implementation detail is that the means of histogram transformation will often influence the choice of data structure used to organise histograms.

Histogram set transformation

The most differentiating factor in sequence models used for compression is the means by which a set of histograms corresponding to a particular context is transformed into a conditional probability.

A common scheme employed is *small-to-large accumulation*, where probabilities of smaller contexts are weighted and combined with probabilities from larger contexts. Similarly, *large-to-small accumulation* specifies the reverse of this process, where probabilities of larger contexts are weighted and combined with probabilities from smaller contexts.

In the former construction, typically the "easiest-to-access/root" histogram is the First Order histogram $*$, which allows for a tail-recursive, depth-first descent through the relevant histogram storage structure. In the latter construction, typically the "easiest-to-access/root" histogram is the most-recently-used, largest-order histogram (*i.e.* $a, b, c, c, *$ from the example in section 15). Histograms are connected through both child-pointers and also *vine-pointers*, which point to the histogram corresponding to the next smaller context. Given vine-pointers, such a method permits a tail-recursive ascent through the relevant histogram storage structure.

[16] The naming convention of "suffix" can confusing when considering that histograms track N-grams which share a common *prefix* to a symbol and a tree structure enables direct access to histograms which contain the next larger *prefix* to a symbol, but compression sequence modelling literature uses suffix framing which will be explained in section 2.1.

An additional scheme sometimes employed is a *cocktail shaker accumulation* where probabilities for all contexts are weighted and combined in a descending, then ascending traversal of the tree (or vice-versa). The descent gathers weighted probabilities of increasingly larger contexts, and the ascent accumulates and weights these probabilities, continually passing the result upwards into a final probability.

Drawing a distinction between these schemes is not only important for designing generic interfaces for efficient implementations of different sequence models, but also gives insight into potential inefficiencies in the cocktail-shaker method. Because the former two methods can be implemented in terms of tail-recursion they do not require dynamic allocation (either explicitly on the heap in an iterative scheme, or implicitly on the stack in a recursive scheme) for increasingly large N-gram contexts. Thus, it can be seen how *traversal* influences histogram set transformation.

To elaborate on how individual probability estimates are combined: most combination schemes are, implicitly or otherwise, depth-dependent and typically weight the probabilities produced from histograms of larger contexts more heavily than probabilities produced from histograms of smaller contexts. Such a combination scheme can almost always be interpreted through a Bayesian lens as adapting probabilities under various priors. Probabilities produced by a single histogram can often be thought of as a transformation of a uniform prior through a Bayesian update with the data contained in the histogram.

Histogram updating

Histogram updating across most sequence models typically takes one of two forms: so called "full-updates" or "shallow-updates". *Full updating* is the simplest choice of updating, where statistics are updated for all applicable histograms for a given context. For example, using the example from section 15 where an a has been observed, the corresponding histograms and N-gram contexts would be updated as follows from biggest to smallest context size:

1. $a, b, c, c, *$: Increment counter for N-gram a, b, c, c, a
2. $b, c, c, *$: Increment counter for N-gram b, c, c, a
3. $c, c, *$: Increment counter for N-gram c, c, a
4. $c, *$: Increment counter for N-gram c, a
5. $*$: Increment counter for N-gram a

Shallow updating (also called *partial updating* or *update exclusion*) is an alternative means of updating counts where only first-time updates are propagated further. Take the same example, but assume that the N-gram c, c, a had been seen before.[17] Under shallow updating, the following procedure would take place:

1. $a, b, c, c, *$: Increment counter for N-gram a, b, c, c, a (first time seen)
2. $b, c, c, *$: Increment counter for N-gram b, c, c, a (first time seen)
3. $c, c, *$: Increment counter for N-gram c, c, a (seen before!)
4. Stop updating! Do not update c, a or a as prior N-gram has been seen before.

This modification means that our statistics count the number of unique contexts in which a given symbol occurred rather than simply the number of times a given symbol has occurred. As described by David J.C. MacKay and Bauman Peto (1995), the former statistic is more powerful in language modelling than the latter. For very (even infinitely!) large N-gram models, shallow updating can improve computational complexity of compression to $\mathcal{O}(N)$ in the message length, compared to $\mathcal{O}(N^2)$ when using full updates.

[17] Necessarily, the N-grams c, a and a would also have been seen before.

Chapter 2

Existing Sequence Modelling Methods

This chapter explores several existing compressors and their associated sequence models. In particular, all of the following models use the same fundamental data structure, a suffix tree, to collect statistics on incoming data.

Discussion will begin with an overview of a suffix tree structure, followed by a description of each compressor and its associated sequence model. A high-level description of how to do *inference*, computation of a probability given symbol observations, and *training*, learning a new symbol, through manipulation of the sequence model's suffix tree will be given as well.

2.1 Suffix Trees

A *suffix tree* is a data structure for efficiently storing and accessing properties related to different suffixes of a string.

Suffix trees are valuable when one views increasing context prefixes of a current symbol as incremental suffixes of the "string" induced by the stream of input symbols seen thus far.[1]

2.1.1 A formal description and demonstration

Suppose a message of length N is being traversed by a compressor, and only L symbols have been seen thus far (thus x_{L+1} is the current symbol

[1] A *prefix trie*, as introduced by De La Briandais (1959) and Fredkin (1960), refers to a similar data structure for storing increasing sized prefixes of a string (for example, in the word "string", the following prefixes are contained $\{s, st, str, stri, strin, string\}$). To conform to the literature's terminology, the term suffix tree will be used.

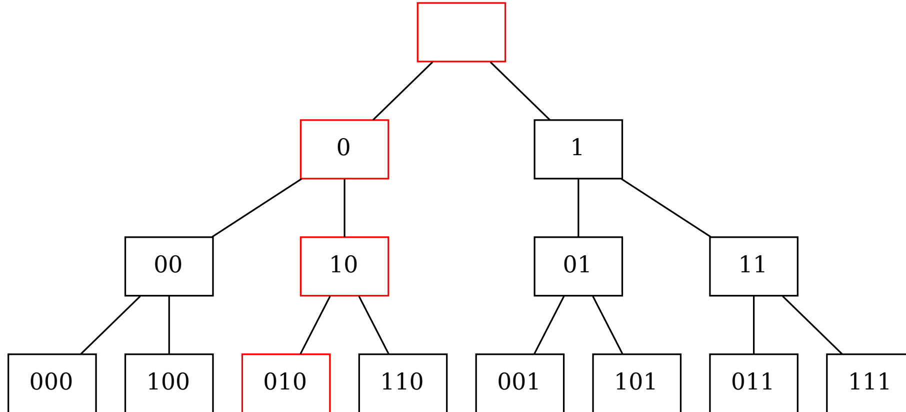


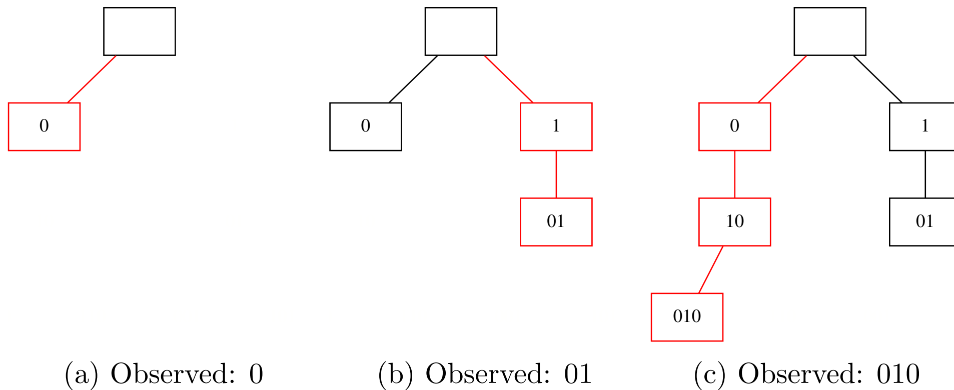
Figure 2.1: A suffix tree of depth $D = 3$ for $\mathcal{A}_s = \{0, 1\}$. Nodes are annotated with the string to which they correspond. Nodes traversed in the search for the partial message 010 are highlighted in red. The root node represents the empty string.

considered by the compressor). The symbols $\langle x_n \rangle_1^L$ form a *partial message*. Many compressors and their associated models need the statistics associated with various suffixes of the current partial message. A data structure which permits efficient access to symbol occurrence statistics of the strings $\{\langle x_l \rangle, \langle x_{l-1}, x_l \rangle, \ldots, \langle x_D, \ldots, x_{l-1}, x_l \rangle\}$ (for some finite depth D), is of immense value, particularly if this access can be provided in an incremental and online manner.

A *suffix tree* is a k-ary tree ($k = |\mathcal{A}_s|$), whose root represents the empty string, and whose nodes contain strings of the same length as their depth. Take $\mathcal{A}_s = \{0, 1\}$. The root node has children corresponding to all length-one strings, and the node corresponding to the string *0* will have children corresponding to all length-two strings ending with *0*, namely $\{00, 10\}$. An illustration of a dense suffix tree is given in Figure 2.1.

Suffix trees can be constructed incrementally, which is demonstrated in Figure 2.2 for the same sequence shown in Figure 2.1. In particular, it can be seen that only partial sequences that have been observed are created, rather than a full, dense representation of all possible N-grams, which would be prohibitively expensive for any large $|\mathcal{A}_s|$ or modestly sized D.[2]

[2] The construction shown here is exactly the lazy histogram allocation scheme introduced in section 1.4.1



(a) Observed: 0 (b) Observed: 01 (c) Observed: 010

Figure 2.2: Incremental construction of a suffix tree for the sequence 010. Red nodes denote nodes that were accessed or created as a result of observing the most recent symbol.

2.1.2 Compacted suffix trees

The aforementioned construction is not optimal in terms of space efficiency. For example, in Figure 2.2b two nodes are created: one for the string 1 and another for the string 01. Similarly, in Figure 2.2c two nodes created: one for the string 10 and another for the string 010. In both of these cases, both created nodes will contain the same information (whatever implementation-dependent details exist for an N-gram with a single occurrence). This duplication can be pathological in a case where every new symbol could create D new nodes (if a novel context is seen in every new symbol).[3]

In both instances, a more spatially efficient implementation might create a *single node* that represents all nodes between the parent node and the new node (inclusive of the new node). For instance, in Figure 2.2b, a single node named "1/01" could be created instead, and similarly in Figure 2.2c, a single node 10/010 could be created. Compressed nodes, described in Fredkin (1960) as "Compact Indicators", still contain all required information, as all nodes compacted in the single renamed node share the same occurrence statistics.

If new node which contains an ancestor in a compacted node needs to be created, the compacted node must be split in order to maintain correctness. For example, if in Figure 2.2b a 1 were to be observed (thus accessing or creating nodes 1, 11, and 011), the compacted node 1/01 would need to be split into 1 and 01, while a new compacted node 11/011 could be formed.

Due to the extra implementation complexity associated with compressed

[3] Such pathological cases are very unusual in human text, but are not uncommon in machine-data.

suffix trees, they were not used in this thesis.

2.2 Context Tree Weighting

Context Tree Weighting (CTW) was introduced in Willems, Shtarkov, and Tjalkens (1995) and described an elegant way of weighting various conditional symbol probabilities of different N-grams through the use of a suffix tree.

2.2.1 CTW's sequence model

The proposed method models a so-called *Bounded-Memory Tree Source*, a generalisation of FSMX sources described in Rissanen (1986).

FSMX sources are a subclass of *Finite-State-Machine* (FSM) sources which model Markov processes.[4]

An *Order-N Markov Process* models the next symbol in a sequence by conditioning *only* on the previous N symbols. In other words:

$$\mathrm{P}\big(x_n \mid \langle x\rangle_1^{n-1}\big) = \mathrm{P}\big(x_n \mid \langle x\rangle_{n-N}^{n-1}\big)$$

CTW uses a suffix tree to access and compute efficiently these N conditional symbol probabilities. In each node, a count is maintained that records the number of times each symbol has been seen with a given context described by that node. Let $\mathcal{C}_c$ be the set of all symbol counts for a particular node corresponding to a context c, let $\mathcal{C}_c[s]$ be the number of times some symbol s has been seen in a context c, and let $|\mathcal{C}_c|$ be the total number of symbols seen in a context c.

[4] The distinction between Bounded-Memory Tree sources, FSMX sources, and FSM sources is subtle. In general, the set of FSM sources is the largest: consisting of any probabilistic model that can be defined in terms of a finite-state-machine. The set of FSMX sources is the smallest: consisting of a probabilistic model that is modelled by a closed suffix tree of parameters. Bounded-Memory Tree sources lie in between the two: consisting of any probabilistic model that can be represented by an arbitrary suffix tree of parameters.

Drilling-down on the definition of *closed*: A tree that contains the suffixes $\{0, 00\}$ is closed (all nodes can be reached from other nodes: namely, 00 can be reached from 0, and 0 can be reached by the empty set). However, a tree that contains the suffixes $\{0, 01\}$ is not closed (01 cannot be reached from 0, it would require a node 1, which does not exist).

CTW formally applies to Bounded-Memory Tree sources, but is practically only applied to closed suffix trees. As a result, the remainder of this thesis will refer to CTW as an FSMX model.

For one of the largest tracked contexts (found at a leaf of the suffix tree), the corresponding conditional symbol probability density $\mathrm{P}\big(x_n \mid \langle x \rangle_{n-N}^{n-1}\big)$ is modelled via a KT-estimator parameterised by α[5] with the expression:

$$\mathrm{P}_{\text{leaf}}(s \mid c) = \frac{\mathcal{C}_c[s] + \frac{1}{\alpha}}{|\mathcal{C}_c| + \frac{|\mathcal{A}_s|}{\alpha}}$$

For interior nodes of some context c, the probabilities of child nodes (which have context denoted by sc, the concatenation of a single symbol s and the parent's context c) are combined and weighted with the interior node's KT-estimator equally:

$$\mathrm{P}_w(s \mid c) = \frac{1}{2} \cdot \mathrm{P}_{\text{leaf}}(s \mid c) + \frac{1}{2} \cdot \prod_{l \in \mathcal{A}_s} \mathrm{P}_w(s \mid lc) \tag{2.1}$$

Note that this definition is a recursive: a child's $\mathrm{P}_w(s \mid lc)$ is also the weighted sum of *its own* leaf probability and *its children's* $\mathrm{P}_w(s \mid l_1 l_2 c)$. For the base case of a leaf node, $\mathrm{P}_w(s \mid c) = \mathrm{P}_{\text{leaf}}(s \mid c)$.

As an effective implementation detail, because $\mathrm{P}_w(s \mid c)$ can be calculated incrementally for $|\mathcal{A}_s| = 2$ (see section A.2), a given node can cache partial calculations. Although it would seem that all nodes are used for every calculation of a conditional symbol probability, because only a single path of a tree is updated after learning any symbol, only a linear number of updates in D must be performed to arrive at a probability.

Thus, computing any conditional symbol probability can be performed in $\mathcal{O}(1)$ (simply consulting the cached $\mathrm{P}_w(s \mid \emptyset)$ in the root node), and updating associated incremental probabilities requires a linear pass from the largest context leaf to the zero-context root.

2.2.2 Inference and learning in CTW

Given the above description of CTW, how does one actually create and incrementally update the given tree?

For some symbol s with an associated context c, begin at the root of the suffix tree, and descend (creating new nodes as needed) down branches according to the symbols of c read backwards. So for example, given $c = \{a, b, d\}$, one should travel down the nodes, in order $\{ROOT, d, bd, abd\}$.

[5] Note that in Willems, Shtarkov, and Tjalkens (1995), $\alpha = |\mathcal{A}_s| = 2$ is used, however, in larger $\mathcal{A}_s$, it was empirically observed in Volf (2002, Section 4.5) that setting $\alpha = |\mathcal{A}_s|$ yields poor performance, and for $|\mathcal{A}_s| = 256$, an $\alpha \in [10, 20)$ yields superior performance over text data.

At the leaf node, increment $\mathcal{C}_c[s]$ and update $\mathrm{P}_w(s \mid c)$. If the parent's context is taken to be $\pi(c)$, then traverse to the leaf node's parent, increment the parent's $\mathcal{C}_{\pi(c)}[s]$ and update the parent's $\mathrm{P}_w(s \mid \pi(c))$ given the child's updated $\mathrm{P}_w(s \mid c)$. Repeat this operation until all traversed nodes have been updated.

2.3 Deplump

Deplump was introduced in Gasthaus, F. Wood, and Teh (2010), which utilises the *Sequence Memoizer* (SM) model described by F. D. Wood et al. (2009). Deplump presents as an explicitly Bayesian approach to compression.

As opposed to CTW, which presents a recursive, autoregressive means of learning the parameters of a tree source used to structure conditional probabilities of upcoming symbols, Deplump is built on marginalising a posterior distribution formed from a non-parametric hierarchical prior.

2.3.1 SM: Deplump's sequence model

The *Sequence Memoizer* utilises a modified Hierarchical Pitman-Yor Process (HPYP). To describe a HPYP, first a Dirichlet Process (DP) will be described in terms of the classic Chinese Restaurant Process construction with a connection to bag-of-words language modelling.

Hierarchical Pitman-Yor Processes

Models that contain the word "process" in their name are typically non-parametric models (which still have hyper-parameters). Furthermore, sampling from a process yields a distribution.

To give a traditional construction describing a DP, which takes a concentration parameter α, consider the *Chinese Restaurant Process* (CRP) whose description is as follows:

Suppose there exists a restaurant with an infinite number of tables with each table having an infinite number of seats, all of which are empty to begin. A customer walks in and sits at the first available table. The next customer that walks in has a probability of $\frac{1}{\alpha+1}$ of sitting at the same table as the first customer, and a probability of $\frac{\alpha}{\alpha+1}$ of sitting at a new table. For any new customer entering a restaurant with each table s having $\mathcal{C}[s]$ customers (for

a total of $|\mathcal{C}|$ customers already present):

$$\begin{aligned} \mathrm{P}(\text{sitting at existing table } s) &= \frac{\mathcal{C}[s]}{\alpha + |\mathcal{C}|} \\ \mathrm{P}(\text{sitting at a new table}) &= \frac{\alpha}{\alpha + |\mathcal{C}|} \end{aligned} \tag{2.2}$$

Advancing this process, suppose that there are now three tables with customers. Suppose that the first table has 5 customers, the second table has 4 customers, and the third table has 1 customer. A new customer who walks in will have the respective probabilities $\frac{5}{10+\alpha}, \frac{4}{10+\alpha}, \frac{1}{10+\alpha}$ of sitting at an existing table, or a probability of $\frac{\alpha}{10+\alpha}$ of sitting at a new table.

Two key properties can be seen from this construction. Firstly, populated tables will become more populated (sometimes termed "the rich get richer"). Secondly, after a large number of customers have entered, it is increasingly unlikely for a new customer to sit at a new table (instead, the customer will likely sit at an existing one).

Connecting CRP with that of a language model, namely sampling from a bag-of-words (or symbols in the case of a compressor's sequence model). A new customer sitting at an existing table corresponds to picking a symbol that has been seen before, and a new customer sitting at a new table corresponds to picking a symbol that has never been seen before. The DP-based bag-of-words model respects the naive intuition that both a common symbol should be seen more often when sampling, and that after many observations, it should be increasingly unlikely to see a new symbol.

However, something that may sit uneasy with those familiar with the power-law properties of natural language is that there are many words which are uncommon as seen in Zipf (1945). Applying this empirically-observed result to our CRP-based-language model, we would like to impose the fact that the likelihood of seeing a new symbol should not decay as steeply as we see more symbols. One way to do so is by incorporating a *discount parameter* d which takes some probability mass from populated tables and assigns it to the probability of forming a new table.

In particular, for a discount parameter $d \in [0, 1)$, concentration parameter $\alpha \in (-d, \infty]$, and the aforementioned three table arrangement, a new customer walking in would have the respective probabilities of $\frac{5-d}{10+\alpha}, \frac{4-d}{10+\alpha}, \frac{1-d}{10+\alpha}$ or a probability of $\frac{\alpha+3 \cdot d}{10+\alpha}$ for sitting at a new table. The probability of forming a new table *decays less quickly* (for $d > 0$) as more tables have been formed. Such behaviour reflects the power-law nature of natural language by nudging our model to create more unique symbols. The addition of this discount parameter to a DP creates a Pitman-Yor Process (PYP).

Extending equation (2.2) for a PYP we have the following probabilities for a new customer entering the restaurant:

$$\mathrm{P}(\text{sitting at existing table } s) = \frac{\mathcal{C}[s] - d}{|\mathcal{C}| + \alpha}$$
$$\mathrm{P}(\text{sitting at a new table}) = \frac{\alpha + \|\mathcal{C}\| \cdot d}{|\mathcal{C}| + \alpha}$$

where $\|\mathcal{C}\|$ is the number of tables that have been created

When applied to sequence modelling, these two constructs can be used to answer the questions: "Will the next symbol I see be novel? If not, which symbol is it likely to be?" Note however that this construction does not utilise any context information, which would be useful to include. We can incorporate context information into the model by introducing a hierarchy to a PYP.

Abandoning the CRP analogy to work with N-grams:[6] suppose that the probability of seeing a particular N-gram composed of context c and a final character s is defined in terms of its *context suffix* $\pi(c)$, for example the 3-gram $\{x, y, z\}$ has $c = \langle x, y\rangle$ and $s = z$ the context suffix is $\pi(c) = \langle y\rangle$, as follows: (Note that $\mathcal{C}_c$ is now specific to a given context c)

$$\mathrm{P}(s \mid c) = \frac{\mathcal{C}_c[s] - d}{|\mathcal{C}_c| + \alpha} + \frac{\alpha + \|\mathcal{C}_c\| \cdot d}{|\mathcal{C}_c| + \alpha} \cdot \mathrm{P}(s \mid \pi(c)) \tag{2.3}$$

The probability associated with an empty context, $c = \emptyset$, is uniform over $\mathcal{A}_s$:

$$\mathrm{P}(s \mid \emptyset) = \frac{1}{|\mathcal{A}_s|}$$

This conditional probability construction of a *Hierarchical Pitman-Yor Process* is recursive and utilises context information. The above recursive definition also fits neatly into the access patterns imposed by a suffix tree.

[6] Note that there do exist similar explanations of hierarchical CRPs, so called *Nested Chinese Restaurant Processes* whose description can be found in Griffiths et al. (2003). However, the description is omitted as the analogy does not intuitively map to the generation of N-grams. To provide a brief extension however, the main intuition which is exploited in the hierarchical extension is that a given customer can sit at one table per restaurant across multiple, distinct, restaurants.

From Hierarchical Pitman-Yor to Deplump

Deplump makes several modifications to the HPYP model described above, which are as follows.

Deplump uses an infinite-depth suffix tree using the suffix tree compaction scheme described in subsection 2.1.2, which corresponds to using N-grams of $N \to \infty$. Additionally, Deplump opts to use depth-specific discount parameters, rather than the global discount parameter implied by equation (2.3). Specifically, if $|c|$ is the size of some context c, Deplump uses unique discount parameters $d_{|c|}$ for $|c| \in [0, 11)$ with $|c| \in [11, \infty)$ all using d_{10}.[7]

Lastly, Deplump makes two modifications to the expression of $\mathrm{P}(s \mid c)$ presented by the HPYP.

First, the next smaller order context's probability should be used if a given symbol has never been seen in the current context. In other words, if $\mathcal{C}_c[s] = 0$, then $\mathrm{P}(s \mid c) = \mathrm{P}(s \mid \pi(c))$. Analytically, this piece-wise property can be expressed through an indicator function of $\mathcal{C}_c[s]$ such that

$$\mathrm{P}(s \mid c)_{UKN} = \frac{\mathcal{C}_c[s] - d_{|c|}}{|\mathcal{C}_c| + \alpha} \cdot \mathbf{1}[\mathcal{C}_c[s] \geq 1] + \frac{\alpha + \|\mathcal{C}_c\| \cdot d_{|c|}}{|\mathcal{C}_c| + \alpha} \cdot \mathrm{P}(s \mid \pi(c)) \quad (2.4)$$

Secondly, Deplump presents two variants, the above method which is termed DeplumpUKN (also called SMUKN), and another, termed Deplump1PF (also called SM1PF). DeplumpUKN corresponds to exactly the same predictive distribution used in Kneser-Ney Smoothing.[8] Because infinitely large N-grams are employed, this technique is called Unbounded Kneser-Ney (hence, DeplumpUKN).

Deplump1PF tracks a second set of per-symbol counts $\mathcal{T}$ which are nondeterministically[9] updated alongside the usual counts $\mathcal{C}$, which are used in the modified predictive:

$$\mathrm{P}(s \mid c)_{1PF} = \frac{\mathcal{C}_c[s] - d_{|c|} \cdot \mathcal{T}_c[s]}{|\mathcal{C}_c| + \alpha} \cdot \mathbf{1}[\mathcal{C}_c[s] \geq 1] + \frac{\alpha + |\mathcal{T}_c| \cdot d_{|c|}}{|\mathcal{C}_c| + \alpha} \cdot \mathrm{P}(s \mid \pi(c)) \quad (2.5)$$

[7] Originally, the SM used only six ($|c| \in [0, 6)$) unique depth parameters.

[8] Kneser-Ney Smoothing is a classic language modelling technique introduced in Kneser and Ney (1995) and summarised alongside other smoothing techniques in Chen and Goodman (1998).

[9] The nondeterminism of $\mathcal{T}$ arises from the fact that Deplump1PF is an efficient implementation of a particle filter of a single particle (hence the abbreviation, 1PF from "One (particle) Particle Filtering") sampling the underlying CRP for each context. Further details can be found in Gasthaus, F. Wood, and Teh (2010, Section 4).

2.3.2 Inference and learning in Deplump

Given these descriptions of different variants of Deplump, how does one construct and build the corresponding suffix tree?

Similar to subsection 2.2.2, for some symbol s with an associated context c, one traverses down suffix tree's branches according to the symbols of c read backwards. Under DeplumpUKN, node counts in $\mathcal{C}[s]$ are updated according to shallow updates described in section 16. Thus, in the "usual" case, some $\mathcal{C}_c[s]$ will not be updated. Under Deplump1PF, the same logic is employed as in DeplumpUKN except shallow updating logic can be overridden if a biased coin lands heads (biased with probability):[10]

$$\frac{d_{|c|} \cdot |\mathcal{T}_c| \cdot \mathrm{P}(s|\pi(c))}{(\mathcal{C}_c[s] - d_{|c|} \cdot \mathcal{T}_c[s]) + (d_{|c|} \cdot |\mathcal{T}_c| \cdot \mathrm{P}(s \mid \pi(c)))} \tag{2.6}$$

If this coin lands heads, $\mathcal{T}_c[s]$ is incremented, and the parent node is updated regardless of the shallow update policy. In such an instance, a new biased coin is flipped in the parent node and the same procedure is followed. If the coin lands tails, $\mathcal{T}_c[s]$ is not changed, and the parent node is updated according to the shallow update policy.

Lastly, a worthwhile observation is that the behaviour of Depump1PF decays to DeplumpUKN if the coin in always lands on tails.

Generating a conditional symbol probability in for Deplump is more complicated than the method described in subsection 2.2.2. In both DeplumpUKN and Deplump1PF, for a symbol s with an associated context c, one traverses the tree downwards according to the symbols of c read backwards, and accumulates results according to either equation (2.4) or equation (2.5) respectively.

2.4 PPM-DP

A common family of algorithms for compression which has produced many practical compressors for a few decades is the *Prediction by Partial Matching* (PPM) family of compressors first introduced by Cleary and Witten (1984).

The following section will consist of a brief explanation of an escape mechanism and blending in the context of PPM, and conclude with a description of PPM-DP.

[10] Note that the expression in equation (2.6) is the proportion of the numerator of the second/"made a new table" term to the sum of the numerators from equation (2.5) (without the concentration parameter α).

Additionally, the $\mathrm{P}(s \mid \pi(c))$ term is calculated *before* updates are applied, meaning that probabilities must be calculated during downwards traversal.

2.4.1 Classical PPM approaches

Though there is some variety and diversity of PPM implementations and techniques, the key concepts are mostly the same. A brief description of these key concepts will be provided below.

Firstly, the general approach of PPM is similar to both CTW and Deplump, where a suffix tree is built incrementally as new symbols from a message are observed. Diverging from the previously mentioned compressors, is how this suffix tree is used to generate conditional symbol probabilities.

PPM generally aims to use only the histogram which exactly matches the longest observed N-gram, but will "back-off" to lower order N-grams if instances of higher order N-grams have not yet been observed (or observed enough times). When this "back-off" is performed, the probability associated with the unobserved, high-order N-gram is attributed to an *escape symbol*, which can have higher mass because it represents *all* hitherto unobserved symbols in a given context. This trait has the computationally beneficial effect of minimising the number node lookups in the suffix tree for the computation of a conditional symbol probability.

Alternatively, some PPM methods use *blending*, which takes a weighted sum of conditional probabilities from relevant histograms for a given symbol's context, similar to CTW and Deplump.

2.4.2 PPM-DP's sequence model

PPM-DP can be thought of as an extension to Deplump and SM, adding additional parameters. In particular, where Deplump only had depth-dependent discount parameters $d_{|c|}$, PPM-DP not only has depth-dependent concentration $\alpha_{|c|}$, but also adds an additional parameter dimension of *fanout*, another term for $\|\mathcal{C}_c\|$, the number of unique symbols seen in some context c. Thus, PPM-DP's concentration and discount parameters are dependent on both depth and fanout: $\left\{\alpha_{|c|,\|\mathcal{C}_c\|}, d_{|c|,\|\mathcal{C}_c\|}\right\}$ which transform equation (2.4) to

$$\mathrm{P}(s \mid c) = \frac{\mathcal{C}_c[s] - d_{|c|,\|\mathcal{C}_c\|}}{|\mathcal{C}_c| + \alpha_{|c|,\|\mathcal{C}_c\|}} \cdot \mathbf{1}[\mathcal{C}_c[s] \geq 1] + \frac{\alpha_{|c|,\|\mathcal{C}_c\|} + \|\mathcal{C}_c\| \cdot d_{|c|,\|\mathcal{C}_c\|}}{|\mathcal{C}_c| + \alpha_{|c|,\|\mathcal{C}_c\|}} \cdot \mathrm{P}(s \mid \pi(c)) \tag{2.7}$$

Note that both $\{\boldsymbol{\alpha}, \boldsymbol{d}\}$ are now matrices of parameters, which are used globally across calculations in the suffix tree. Steinruecken, Ghahramani, and David MacKay (2015) find that a good configuration for human text uses $\{\boldsymbol{\alpha}, \boldsymbol{d}\}$ of sizes $|c| \in [0, 8)$ and $\|\mathcal{C}_c\| \in [0, 13)$ in a configuration dubbed

"N8", with larger values of either depth or fanout "clamping" to the largest available parameter configuration. For example, $d_{10,100} = d_{7,12}$.

The motivation for adding an additional dimension of parameters is to attempt to provide different weightings for different classes of contexts. The authors find that fanout and depth are sufficient to define and distinguish different classes of contexts for the purposes of improving compression performance on human-readable text.

This quadratic growth in parameters presents potential difficulties in finding their appropriate values. As its name suggests, PPM-DP approaches this challenge through the use of dynamic parameter updates. PPM-DP takes a gradient-based approach to determine appropriate values for $\{\boldsymbol{\alpha}, \boldsymbol{d}\}$. In particular, the objective to be minimised is compressed sequence length, which (as described in subsection 1.2.2) is a function of the information content of a sequence: $-\log_2 \mathrm{P}(\langle x \rangle_1^n)$. Due to properties of logarithms and the factorisation of conditional probabilities:

$$\nabla - \log_2 \mathrm{P}\left(\langle x \rangle_1^N\right) = \nabla \sum_{n=1}^{N} - \log_2 \mathrm{P}\left(x_n \mid \langle x \rangle_1^{n-1}\right)$$

Given that our model identifies $\mathrm{P}\left(x_n \mid \langle x \rangle_1^{n-1}\right) = \mathrm{P}(s \mid c)$ given in equation (2.7), incremental gradients of $\frac{\delta \mathrm{P}(s \mid c)}{\delta \alpha_{|c|, \|\mathcal{C}_c\|}}, \frac{\delta \mathrm{P}(s \mid c)}{\delta d_{|c|, \|\mathcal{C}_c\|}}$ can be calculated and used to update parameters appropriately either in an online or offline manner.

2.4.3 Inference and learning in PPM-DP

Tree construction is identical to DeplumpUKN found in subsection 2.3.2, and either shallow or full updates can be employed. If online parameter updates are used, the respective gradient updates are applied before counts are updated and can be incorporated in the same upwards traversal that is used when learning a symbol.

Chapter 3

A *Contaminating* Compressor

This chapter will explore the use of *contamination* in compression as a proposed modification to the existing compressors described in chapter 2. A *contaminating* compressor utilises a fixed-size hash table as the backing storage of a sequence model's suffix tree, which permits (and embraces) hash collisions for memory and time efficiency. This chapter will begin with a brief overview of hash tables, followed by a brief overview of the sparse literature concerning constant-runtime-memory compressors, and then a smaller overview of attempts in the literature to apply *contamination*. Finally, we will analyse results of applying *contamination* to existing compressors, with a focus on comparing compression effectiveness against run-time memory usage.

3.1 Hash Tables

A hash table is a family of data structures for efficiently storing associative information, where a *key* is associated with some *value*. Specifically, some *hash function* h maps a key to an index into a table, or series of buckets, where an associated value can be found. *Hash collisions* are when multiple distinct keys map to the same index. In the event of a collision, some form of *collision resolution* is typically employed. Additionally *rehashing*, or changing the number of buckets, typically according to the *occupancy* or *load factor* of the hash table, can help to minimise the number of collisions. In general, for a good rehashing criterion and a good hash function, hash tables can be advantageous over trees in terms of retrieving associative data because a hash is typically computed in $\mathcal{O}(1)$ whereas tree traversal is typically $\mathcal{O}(\log N)$ in the number of keys stored thus far.

Though there are a diverse number of techniques for *collision resolution*

and *rehashing*, none are of particular interest to this thesis because *contamination* *embraces* collisions.

Hash tables, as used in a *contaminating* compressor, have the following weakened conditions which greatly simplifies their implementation. Firstly, the hash table is of a fixed-size, meaning that managing rehashing for table expansion is no longer required. Secondly, the hash table permits collisions. As a result, complex collision resolution schemes need not be considered (though having an understanding of the dynamic load factor, and the general distribution of keys throughout the hash table will still be valuable). Lastly, the hash table does not need to accommodate individual deletions, just successful destruction. Consequently, node deletion, a complicating implementation detail in many hash tables, can be ignored.

3.2 Existing Work

3.2.1 Constant-Size compressors

All practical implementations of compressors have an implicit, constant memory limit, which is the total available memory available to the compression system. Though many workstation systems can silently increase this limit via paging and memory compression (as shown in Iyigun and Juarez (2015)), many embedded systems do not have this luxury.

Unbounded, dynamic growth of data structures for compression is problematic in many applications. As a result, there is usual a desire for some method of imposing constant runtime memory constraints on compressors. One popular approach is *Amnesia*, which deletes a compressor's entire backing data structure once a pre-defined memory cap is reached. Another popular approach is *Random Deletion*, which deletes random nodes in the backing data structure either once a memory cap is reached, or during insertion.[1] Additionally, techniques which interface with a compressor's underlying sequence model can be employed, and generally follow the strategy of heuristically deleting portions of the backing data structure that are deemed less useful in making good compression decisions. For example, Bartlett, Pfau, and F. Wood (2010) use a heuristic termed *greedy deletion* to remove nodes which cause minimal disturbance to the likelihood of the observed sequence. The authors found, however, that this heuristic-guided deletion scheme only offered marginal improvement over a uniform deletion scheme imposed by

[1] See McFadden (1992, Lesson 5, Lesson 9), which discusses the implications of Random Deletion and Amensia respectively in early adaptive compressors.

Random Deletion. However both schemes were demonstrably more effective than Amnesia.

3.2.2 Hashing compressors

Hash tables have often been used as a backing data structure for efficient implementations in tree-based sequence models, see Volf (2002, Section 4.4) as well as Cleary and Darragh (1984), Franken and Peeters (2003), and C. Bloom (2010). However, there is little documented work on the use of hash table-based compressors which are explicitly designed to permit collisions. To the best of our knowledge, the only instances of such cases are as follows.

Lelewer and Hirschberg (1991) discuss several performance-motivated modifications to PPMC, among which is having all third order context histograms use a double hashing scheme for partial collision resolution. Specifically, a context string will be hashed and looked up in a table, if the corresponding entry is occupied but not by the same context string (as determined by a separately generated parity byte), a second hashing scheme is used, which when looked up will have an unconditionally updated entry potentially "trampling" the existing contents.

Howard and Vitter (1992) mention that "Hashed high-order Markov models", (single hashing and no collision resolution) do not significantly degrade compression performance as one might expect. Although no compression effectiveness metrics are given, the authors do make a valuable passing remark that in the "worst-case" of a large-number of contexts sharing a single bucket, the heavily collided bucket's histogram will approach the characteristics of a 0-order histogram over the same data seen thus far.

Rein, Guhmann, and Fitzek (2006) discuss the design of an ultra-low memory (128 kilobyte) compressor which also takes the form of PPMC and utilises single hashing with so-called *collision avoidance*. Collision avoidance in this work means that if a context's hash maps to an existing, non-matching entry, the statistics from a lower-order context are used instead. The authors note that disabling collision avoidance (*i.e.* *contamination*) results in poor compression.

Lastly, Mahoney (2005)'s PAQ family of compressors implicitly permit a modest number of collisions through the use of a relatively lax means collision detection,[2] as a means of achieving low-latency hash-table element accesses

[2] See `paq1.cpp::HashElement`, which uses an 8-bit checksum for context uniqueness. Over $|\mathcal{A}_s| = 256$ and a context depth of 3, this yields a reasonable amount of collisions. Furthermore, in PAQ7 (see `paq7.cpp`'s "IMPLEMENTATION" comment), this checksum was removed entirely for high-order contexts, using a weaker form of collision detection instead.

by reducing cache misses.

We can see in the existing literature, that all preliminary work in this area exclusively uses hashing as a means of dealing with histograms of high-order contexts, and use alternative structures for managing histograms of low-order contexts. Furthermore, we find conflicting reports about the benefit of *contamination*, with some authors reporting negligible loss in compression effectiveness and others reporting catastrophic degradation.

3.3 Random Hashing

The first attempt to examine the characteristics of a *contaminating* compressor was using a hash table with no collision resolution, and variably-sized tables, applied directly to the existing methods described in chapter 2.

All aforementioned methods were implemented with a fixed context depth $D = 8$, and allowed to execute with an unbounded amount of memory for each file in the corpus. Separately, a hash-variant of each technique was deployed to compress the same files corpus files with varying amounts of memory. Note that the unit of memory used is a *Histogram*, which have been represented by an 8-bit counter for every symbol in the alphabet (rescaling follows the procedure described in section A.1), with two extra numbers representing the total number of symbols seen in the context, $|\mathcal{C}_c|$ (32-bit counter) and the number of unique symbols seen in the context $\|\mathcal{C}_c\|$ (8-bit counter). As a result, a Histogram's size is

$$8 \text{ bits} \cdot |\mathcal{A}_s| + 32 \text{ bits} + 8 \text{ bits}$$

For $|\mathcal{A}_s| = 256 + 1$ (An 8-bit alphabet with an out-of-alphabet EOF symbol) a Histogram is 262 bytes big. Figure 3.1 shows the results of an initial experiment that uses a uniform random hash table as the backing store for the suffix tree methods presented in chapter 2 over the Calgary corpus. In particular, the hashing algorithm used is boost's `hash_combine`, which is described further in section A.3.

To construct the hashes of increasing-sized contexts, first the zero-order context's hash is taken. This hash is then combined with the most-recent symbol to form the first-order context hash. The first-order context hash is combined with the second-most-recent symbol to form the second-order context hash. This process is repeated until the maximum context depth is reached. The hashes are then converted to table indices by taking the modulus of the hash with the allocated table size.

To provide a better intuition for these early results, firstly, one can note that the baseline compression methods CTW, SMUKN, and PPMDP follow

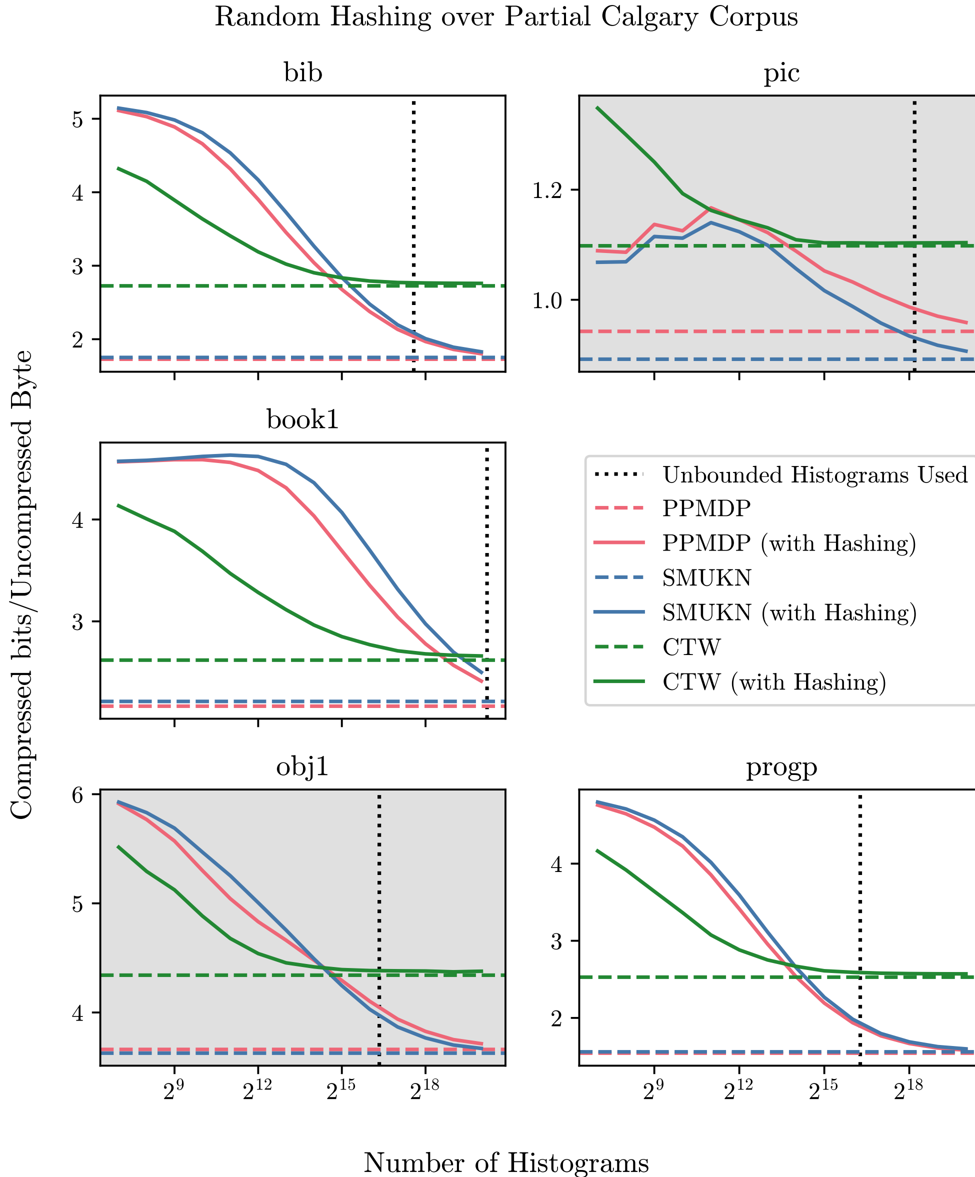


Figure 3.1: Comparison of standard compression schemes all using a finite context depth of $D = 8$ with associated *contaminating* variants over select files in the Calgary corpus. Dotted vertical line indicate memory usage and compression ratio of unbounded scheme. Grey figures denote that the test file is not human-readable text. Data points are sampled for every 2^p for integer $p \in [7, 21)$. SM1PF is omitted as it exhibits extremely similar compression effectiveness to SMUKN.

known results on human-readable text files: power-law-based compressors such as PPMDP and SM variants tend offer superior compression effectiveness over FSMX-based compressors such as CTW. Secondly, for human-readable text, compression effectiveness improves for hash systems which have more memory, ultimately approaching the performance of their respective unbounded variants.

One point of practical interest in these plots is the compression effectiveness where the hash-based compressors have less memory than the unbounded compressors. Though, it can be of passing interest, especially in the case of power-law-based compressors, to see under what memory restrictions a Hash-based scheme approaches the compression effectiveness of the corresponding unbounded scheme. Note that for the FSMX-based compressors, compression effectiveness of the hash variant appears to be identical to the unbounded variant once equivalent number of histograms are provided. However, for power-law-based compressors, it appears that *more* memory must be given than the unbounded variant to attain identical compression effectiveness.

Additionally across all human-readable text files, there is consistent similar compression effectiveness between both power-law compressors SMUKN and PPMDP, which tend to have a quasi-convex characteristic, which is distinct from the quasi-concave characteristic presented by the FSMX-based compressor CTW. A consequence of these concavity characteristics is that although unbounded CTW has poorer compression effectiveness than unbounded SMUKN and PPMDP, HashCTW has *better* compression effectiveness than Hash-Power-Law compressors for stronger memory constraints.

One potential explanation for these different characteristics is the fragility of power-law compressors to corrupted histograms. Specifically, from equation (2.5) and equation (2.7), such compressors rely on correct distinction between zero and non-zero counts of observed symbols in distinct contexts (note how the activation of $\mathbf{1}[\mathcal{C}_c[s] \geq 1]$ drastically changes the expression of $\mathrm{P}(s \mid c)$).

This explanation is substantiated when analysing the behaviour of such systems with Amnesia constraints applied, which can be found in section 3.4.

As a curiosity, it is worth commenting on the behaviour of these compression schemes on binary data files. For geo, obj1, and obj2, the rough pattern seen in human readable text appears to be followed, where the HashFSMX-compressor appears to have a better compression ratio than Hash-Power-Law for most of the domain of interest. Note however, that pic demonstrates peculiar properties of our compressors. For the HashFSMX compressor, the characteristic curve appears similar to human-text files. However, for the power-law compressors, compression effectiveness appears to *worsen* initially

given more memory, and then improve only after some threshold.

This behaviour too, will be clarified when analysing the behaviour of Amnesia variants of these systems.

Figure 3.2 shows the compression results of these compressors on the Canterbury corpus, which consists of larger files than the Calgary corpus. We can see the same broad behaviour as for the Calgary corpus with a few nuanced differences. Over most text files, the HashFSMX variant no longer achieves nearly identical effectiveness to its unbounded counterpart once equivalent memory is allocated. Notably, this is not due to a poor choice of hash function (and using alternative, non-cryptographic hash function, such as FNV,[3] does not change this behaviour).

As with the Calgary corpus, binary data files exhibit some peculiar characteristics of our compressors. For example, both sum and kennedy.xls amplify the HashFSMX compressor's behaviour of worsening compression effectiveness given more permitted memory after a certain threshold.

3.4 Comparison with Amnesia

Amnesia is the concept of destroying all accumulated statistics once a memory threshold has been reached. This differs from *contamination* because Amnesia maintains *purity* of occurrence statistics up to a memory cap, whereas *contamination* opts to capture and preserve more unique context statistics at the cost of losing purity. An interesting question of this work is whether purity the most efficient use of memory. From Figure 3.3, several properties are immediately apparent which might help answer this question.

For human-readable text, it can be seen that all HashCTW models outperform AmnesiaCTW models. This result suggests that for FSMX-based compressors, maintaining histogram purity is not a good use of memory. Contrarily, it can be seen that for many files AmnesiaPPM-DP outperforms HashPPM-DP. This demonstrates that maintaining histogram purity permits PPM-DP to have better compression effectiveness than maintaining more histograms that may be corrupted.

As an aside, the curious results of HashPPM-DP on pic are heightened given the comparison to AmnesiaPPM-DP. The most effective compression appears to be presented by AmnesiaPPM-DP with mid-range allocation of memory as demonstrated by a minimum around 2^{13} histograms. Likely, this owes to the nature of pic itself, a fax image with much blank space (seen in Bell et al. (2000)), having drastic changes in distributional statistics after a

[3] The impact of hash function choice on compression effectiveness is further investigated in section A.3.

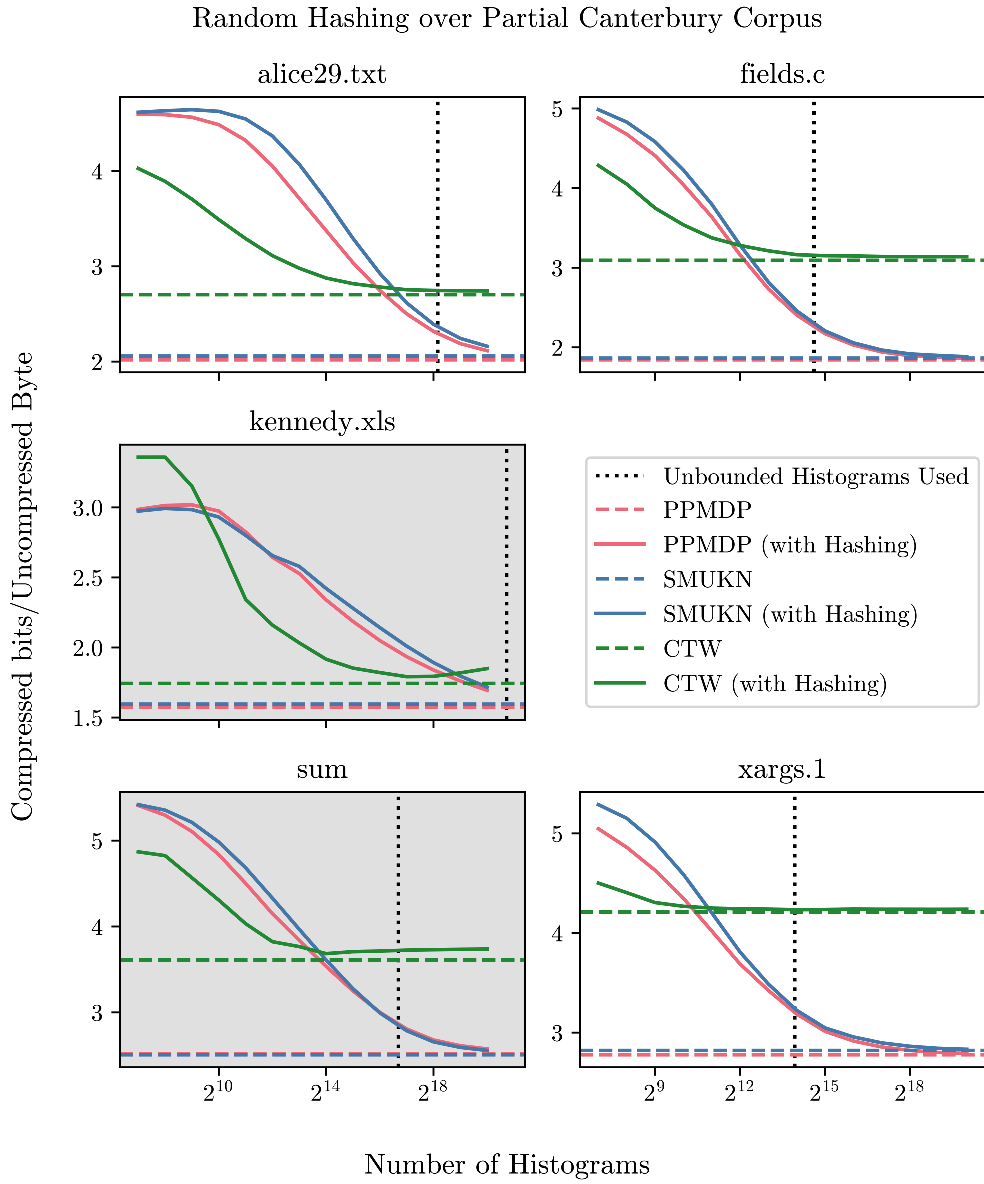


Figure 3.2: Comparison of standard compression schemes with *contaminating* variants over select files Canterbury corpus.

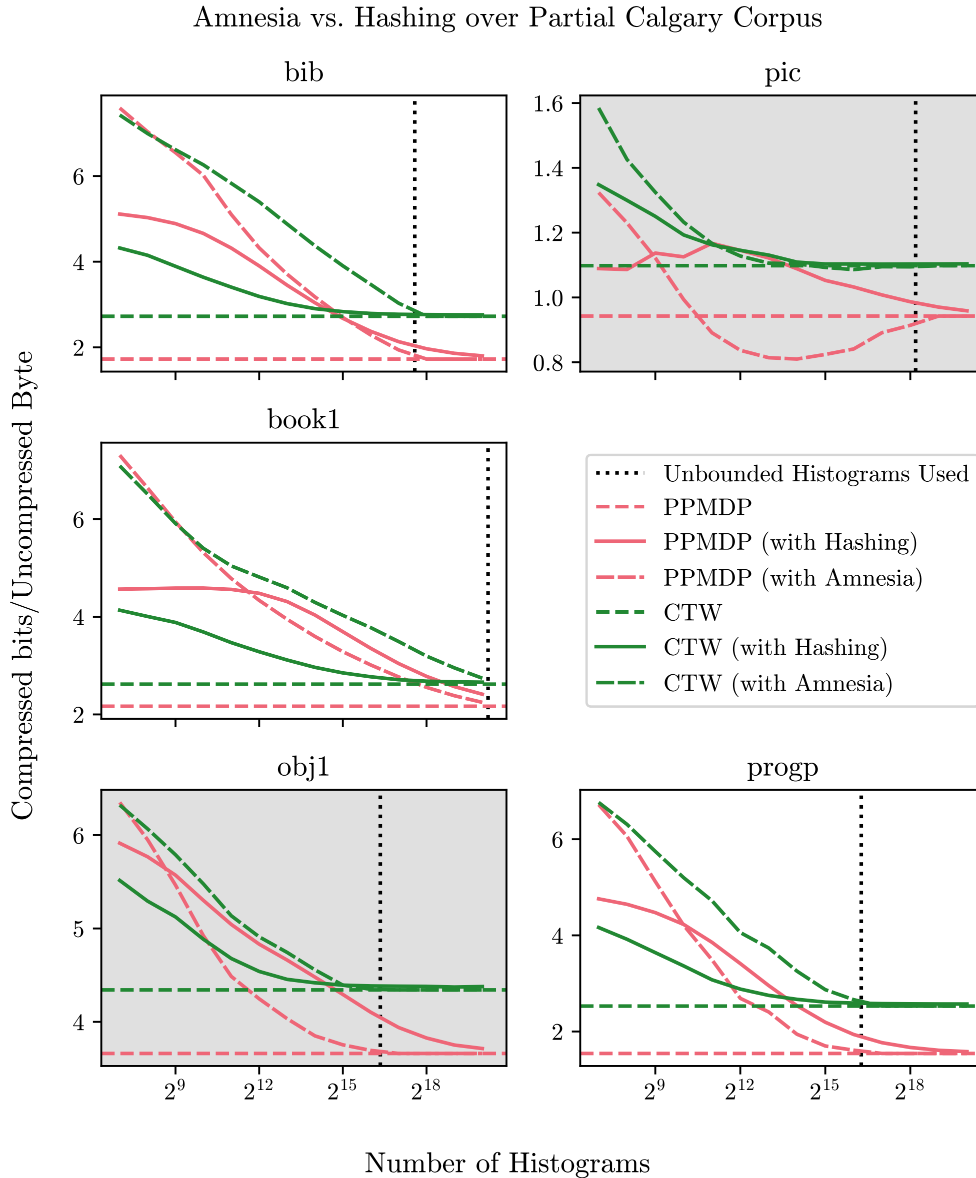


Figure 3.3: Comparison of PPM-DP and CTW with *contamination* and Amnesia variants over select files in the Calgary corpus. For Amnesia, the *Number of Histograms* represents the maximum number of Histograms the compressor is allowed to create, beyond which the backing Suffix Tree will be destroyed. SMUKN is omitted due to exhibiting similar behaviour as PPM-DP.

fairly long runs of consistent data. The rapid adaptability bestowed by the imposition of tight memory constraints may also explain the degradation in compression effectiveness when Hash-Power-Law compressors are given more memory in Figure 3.1.

3.5 Maintaining Pure Histograms

The existing work on *contaminating* compressors discussed in subsection 3.2.2 have predominantly used hashing for histograms of high-order contexts. This decision likely stems from a confluence of factors, among them intuition (histograms of low-order contexts are more influential in weighting, so they should be kept pure; histograms of high-order contexts are already less informative due to sparsity, thus there should be little harm in their *contamination*) and ease-of-implementation (many implementations in the PPM family of compressors explicitly use back-off, which may lend itself more easily to using different computational structures to represent different context orders of histograms). It is thus worthwhile to see the effects of maintaining pure low-order contexts on compression effectiveness.

The strategy used to ensure that certain context depth's histograms remained pure was to divide the flat storage used for hashing into two portions. One portion follows the usual implementation of a dense k-ary tree in a linear array, while the second portion follows the uniform hashing scheme described in section 3.3. Histograms context depths up to and including D_p are tracked in the first portion, while histograms for larger context depths are tracked in the second portion. Only systems that are large enough to contain enough elements such that the first portion is full and the second portion is non-empty were examined in experiments.

As demonstrated in Figure 3.4, maintaining purity of histograms corresponding to lower-order contexts appears to have a negligible impact on compression effectiveness. For histogram allocations which are "just barely" large enough (the integer power of two greater than $|\mathcal{A}_s|^{D_p}$), compression effectiveness appears to be *worse*. This effect can be clearly seen in the protrusion at 2^{17} in book1 and book2. Relating to the discussion about effective use of memory in section 3.4, it appears that maintaining low-order histogram purity is not an efficient utilisation of memory.

One potential rationale for such behaviour is that the occurrence counts of low-order N-grams are sufficiently dense and large that they are robust to small perturbations induced by corruption with sparser and smaller high-order N-gram counts. Furthermore, allocating less overall capacity to capture high-order N-gram statistics yields a less effective compressor.

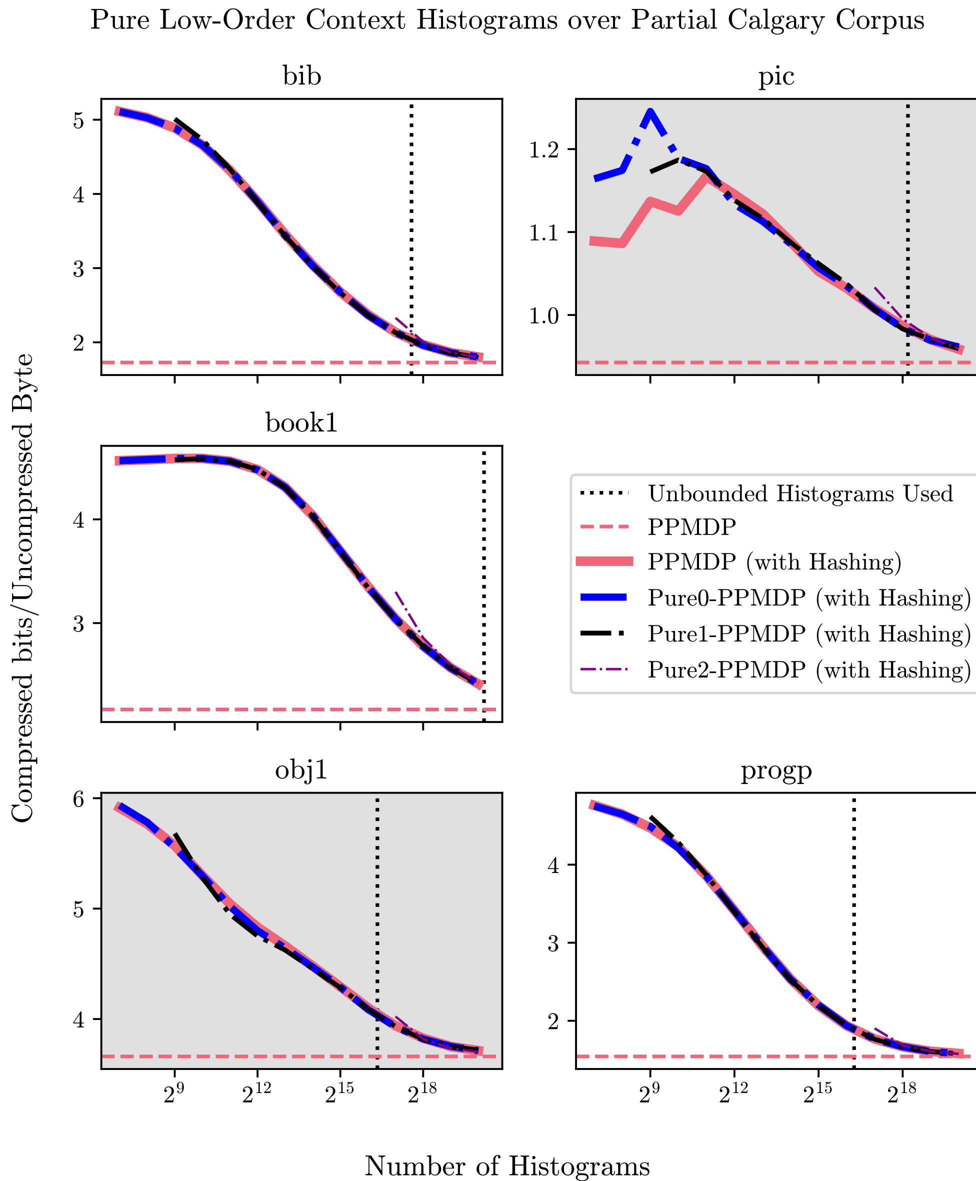


Figure 3.4: Demonstration of HashPPM-DP with various amounts of reserved “pure” histograms. CTW and SMUKN are omitted due to exhibiting similar behaviour as PPM-DP in all cases. Decreasing line widths have been used to demonstrate how similarly all systems behave.

3.6 The Influence of Full Updates

As presented in section 16, full updating is the choice of updating the statistics of all relevant histograms for a given context observed. Full updates contrast with the more widely used scheme of shallow updating, which only updates histograms for which a given context is newly observed. In a similar sense of attempting various techniques found in the literature, perhaps examining full updating is a worthwhile avenue?

As seen in Figure 3.5, using full updates appears to consistently provide worse compression effectiveness in *contaminating* variants (with the exception of pic). The difference in effectiveness for a single file appears to be consistent for most of the memory allocation domain, and is roughly equal to the effectiveness difference between the unbounded variants.

Likely, this constant offset owes to the PPM-DPFull variant using the same depth and fanout-dependent parameters as the PPM-DP-shallow variant, which were tuned specifically for shallow updates. This margin could likely be narrowed by re-tuning the parameters for a system with full updates.

3.7 Concrete Trade-offs in Compression Effectiveness

Throughout this section have been numerous plots of compression effectiveness against memory usage in a logarithmic scale. Presenting these results in a linear scale provides a more intuitive scheme to reason about the trade-off between memory usage and compression effectiveness.

A novel metric to analyse this trade-off in a linear scale is *Compression Proportion* or *CP*, which is defined in terms of a baseline B and candidate C such that:

$$CP_{\mathrm{B}}(\mathrm{C}) = 2 - \frac{\mathrm{C}}{\mathrm{B}} = 1 - \frac{\mathrm{C} - \mathrm{B}}{\mathrm{B}}$$

In the event of comparing quantities where *lower is better* and $\mathrm{C} \geq \mathrm{B}$,[4] Compression Proportion is a visually appealing metric because *higher values are better* and the best score is one.

From Figure 3.6, the *contaminating* variant of CTW outperforms all other constant-space variants for low memory usage, while Amnesia variants achieve better compression effectiveness for memory usage around 45% of the baseline unbounded case. A key takeaway is that HashCTW achieves

[4] Both compression ratio and information content under a given model (when using an unbounded compressor as a baseline and a constant-memory variant of that compressor as a candidate) obey these conditions.

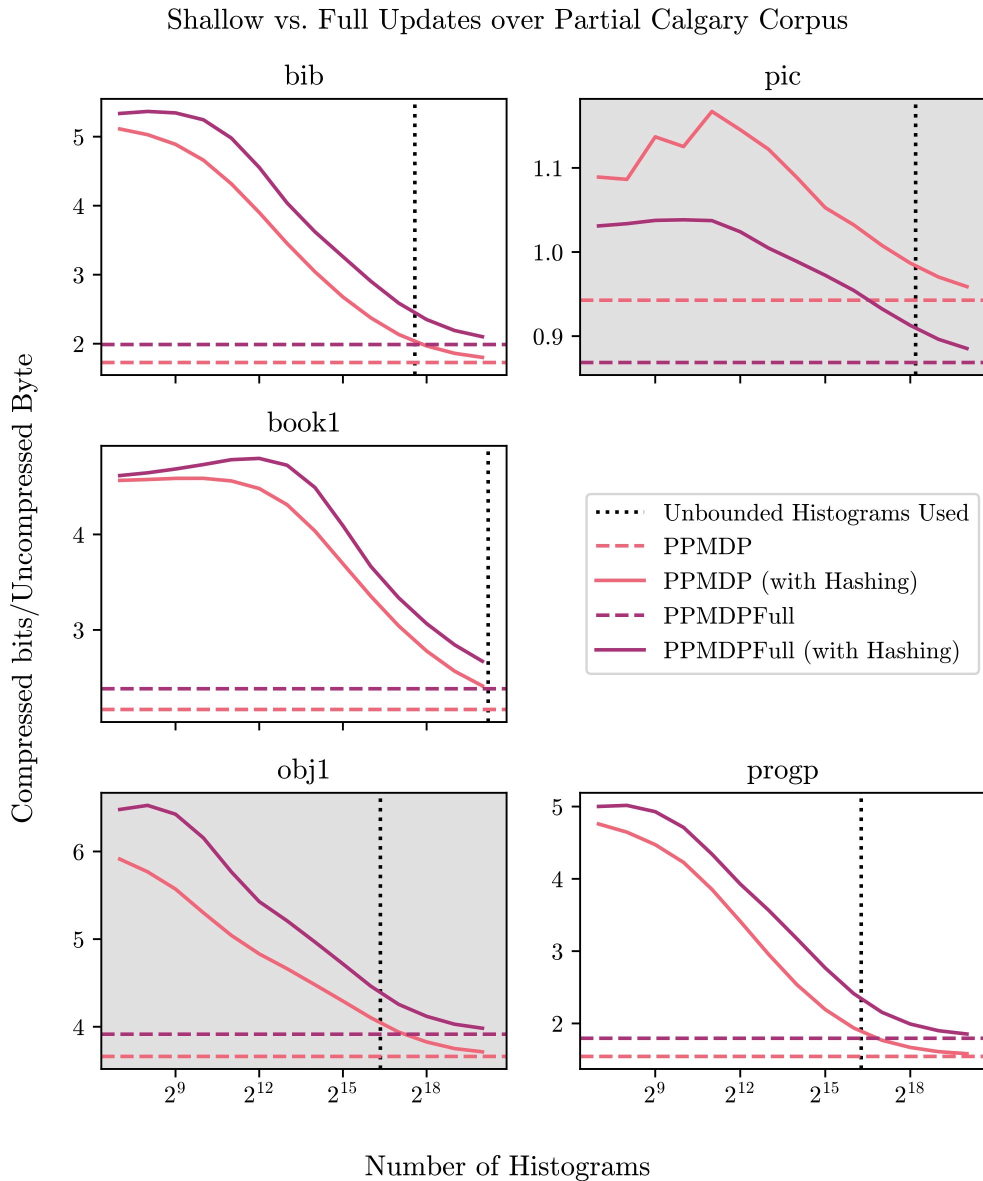


Figure 3.5: Demonstration of HashPPM-DP and HashPPM-DPFull variants over the Calgary corpus. CTW omitted because it inherently uses full updates. SM variants omitted because they present near identical behaviour to PPM-DP.

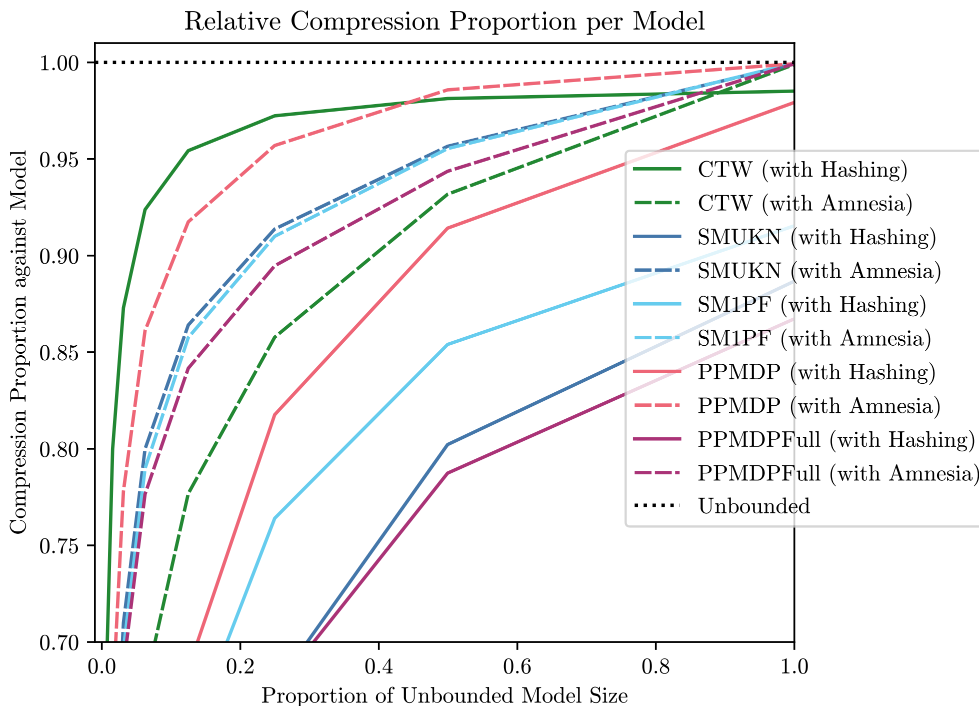


Figure 3.6: Compression proportion of variants relative to their unbounded baseline models. For example, the HashCTW curve describes CP_{CTW}(HashCTW).
Higher values (similar compression effectiveness to the baseline) that are to the left (using less memory) are better.
The test file used is the entire works of Shakespeare concatenated (5,736,236 bytes). All unbounded baseline models required 4,200,496 histograms.

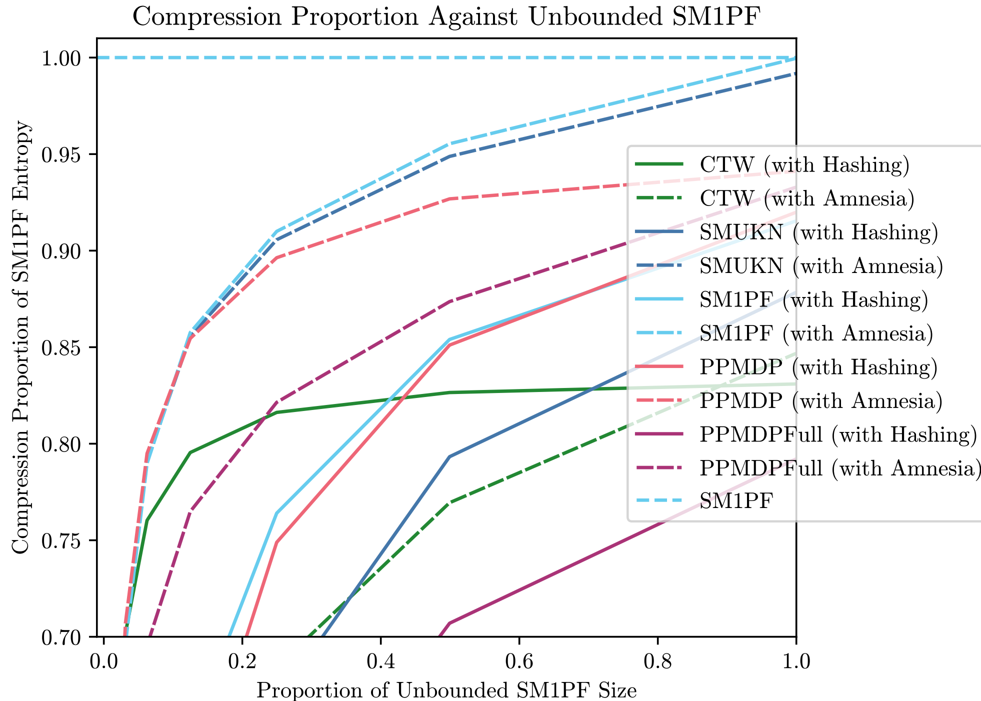


Figure 3.7: Compression proportion of variants relative to the best performing model, unbounded SM1PF with a compression ratio of 1.899 $\frac{b}{B}$. For example, the HashCTW curve describes $CP_{\text{SM1PF}}(\text{HashCTW})$

only a 2.5% worse compression effectiveness using only 30% of the memory as its unbounded variant.

Though, Figure 3.6 is misleading, as the baseline is variable and is thus not against the best model. Figure 3.7 paints a different picture of the compression effectiveness landscape more in line with results shown in Figure 3.3. Explicitly, it appears that Amnesia-Power-Law variants present better compression effectiveness than all *contaminating* variants for any memory allocations greater than around 20% of unbounded memory usage.

3.8 Implementation Notes

3.8.1 Implementation of Amnesia

For Amnesia, a relatively straightforward approach was taken where *after* a symbol has been learned, evaluation of the model's memory footprint takes

place. If the memory footprint is found to exceed the given threshold, the backing data structure is destroyed and the symbol is re-learned in the given context. A subtle hitch is that if the context depth is sufficiently large, and the memory cap is sufficiently small, this can result in an infinite loop.[5] In reality, this "check-after learning" approach is unrealistic, as running out of memory would typically happen *during* the course of a symbol being learned. Real implementations on embedded devices would not be overwhelmingly more complicated, as one could have the out-of-memory handler trigger the destruction of the context tree and re-learn the current symbol.

3.8.2 Implementation notes for suffix tree compressors

There were a number of important design considerations when implementing the compressors for this thesis. The priorities in implementation were the maintenance of correctness, reasonable performance, and rapid development to facilitate searching through a wide number of different techniques. Below are some relevant design aspects of this thesis's supporting implementation which were helpful to address these priorities.

Distinguishing between integer-encoded entities

Most compressors need to deal with integer-encoded data for several distinct entities. These entities include input symbols, output symbols, input alphabet size, output alphabet size, symbol counts, and histogram bookkeeping (such as pointers or references to histograms in a backing data structure). Because many C-like programming languages support generally lax integer conversion,[6] improper differentiation between these entities can lead to extraordinarily subtle bugs. Although proper programming discipline (*i.e.* not making mistakes) and a plethora of runtime checks are approaches that work for many compressor implementations in either a practical or research context, the following insight was valuable in this thesis's supporting implementation.

A sequence model need only be responsible for understanding characteristics of the input symbol alphabet. Because the input symbol alphabet can be described statically (*i.e.* in terms of permitted values and a total size), and

[5] For precisely this reason Figure 3.3 has a lower bound of permitted histograms much larger than the employed context depth.

[6] In C and C++, these conversions are referred to as the *Usual arithmetic conversions* (described by *Implicit Conversions* (2022)), in Java, they are *Integer Conversions and Promotions* (described by Gosling et al. (2013)), and in C# these conversions are referred to as the *Built-in Numeric Conversions* (described by Wagner et al. (2022)).

the declaration of a sequence model can be made dependent on such a static definition, in a programming language with sufficient compile-time programming facilities it is possible to form a compile-time error when one attempts to perform any invalid conversions implicitly. For example, a program attempting to compress bit-symbols with a compressor declared to compress byte-level symbols will not compile successfully. Moving such correctness-testing to compile-time drastically improves run-time performance, as run-time symbol checks are no longer necessary to ensure correctness.

Similarly, all other aforementioned entities can be strongly-typed[7] which will create compile-time errors for any invalid conversions between such entities.

Separating traversal and histogram manipulation

A typical implementation of a suffix tree compressor embeds child and vine pointers in the histogram and marries traversal logic with histogram manipulation. In the typical implementation of a hash compressor it is more natural, however, to separate node traversal from histogram manipulation. This is because histogram lookup (typically through a hash function) is done globally in a single table, rather than locally traversing histograms (*a.k.a* "pointer-chasing"). If this externalised structure is mimicked in the unbounded/suffix-tree implementations, then histogram manipulation need only be written once for each method, and traversal logic can be implemented separately. Such modularisation schemes are important because compressor implementations are often subtle, complex, and error prone. Additionally, if modularisation is achieved, then for N unique traversal methods and M unique histogram manipulation methods, only $N + M$ methods must be implemented rather than $N \cdot M$ methods.

As a result, the dominant implementation structure of suffix tree compressors in this thesis's supporting implementation was to maintain a flat storage of histograms with an external adjacency list describing node mappings. Such a design provides clean separation of histogram manipulation and traversal logic.

Composition through static polymorphism

The aforementioned modularisation and separation of concerns permit a declarative means of defining novel compressors via composition.

[7] In C, for example, any unique `enum` is strongly-typed, meaning it cannot be implicitly converted to any other type. In other C-like languages, the use of an aggregate such as a `class` or `struct` is a more idiomatic way to produce strongly-typed integers.

For example, the traversal methods described in section 16: `TopDownTraversal`, `BottomUpTraversal`, and `CocktailShakerTraversal` can be combined with different backing schemes, such as `AdjacencyList` or `Hashing`.[8] This composition can be extend further, for example, by the `Hashing` backing scheme relying on `FNVHash` or `hash_combine`. To illustrate a full-featured declaration of a compressor using C++ template syntax, the traversal scheme of a CTW compressor utilising FNV hashing could instantiated declaratively as `TopDownTraversal<Hashing<FNVHash>>`.

Though the notion of clean interface separation is not novel in the context of research compressor implementations, existing implementations realise composition through *dynamic polymorphism*. For example, `TopDownTraversal` would maintain a `virtual` reference to some arbitrary backing scheme. The runtime costs of dynamic polymorphism (typically an indirection through a virtual table) are nontrivial when dereferencing is on the "hot path", which is the case for compressors, as operations are performed per-symbol, symbols are bytes, and input files are typically millions or billions of bytes. However, the use of static polymorphism (achieved in this thesis's supporting implementation through the use of C++'s template facilities), permits the compiler to "optimise away" the costs of dynamic polymorphism while keeping the benefits of readability and single-definitions.

[8] Distinct histogram manipulation schemes are necessarily attached to a corresponding traversal scheme. For example, although one could associate a CTW manipulation scheme (which relies on a top-down traversal) with a bottom-up traversal, one would not arrive at coherent probabilities because probability weighting inherently favors lower-order contexts. The same applies to PPM-DP, which could theoretically function if depth and fanout-dependent parameters were re-trained.

Chapter 4

Conclusion

This thesis explored a novel, generally-applicable technique to sequence models used in lossless data compression: the use of *contamination* in suffix-tree-based sequence models to impose $\mathcal{O}(1)$ memory usage restrictions. A *contaminating* compressor utilises a fixed-size hash table as the backing storage of a sequence model's suffix tree, which permits (and embraces) hash collisions for memory and time efficiency. In some cases, *contamination* presents improved utilisation of reduced memory over other generally-applicable techniques, such as Amnesia. For appropriate choices of parameters, *contamination* was found to reduce peak run-time memory usage of certain sequence models by 70% at a cost of only 2.5% worse compression effectiveness.

This thesis can make the following recommendations for the design of a suffix-tree-based compressor that operates on human-readable text. For intense memory restrictions (using less than 1% of the peak run-time memory usage of an unbounded suffix-tree-based compressor), one should use *contamination* over Amnesia as a means of restricting sequence model size (regardless of the underlying sequence model). With slightly weaker memory restrictions (anywhere from 10% or more of the peak run-time memory usage of an unbounded compressor) and a sequence model that is power-law-based, one should use of Amnesia over *contamination* as a means of restricting sequence model size. In the case of a compressor whose underlying sequence model is FSMX-based, one should use *contamination* over Amnesia.

A difficult design decision in the implementation of a compressor is determining a good trade-off between resource utilisation and compression effectiveness. The tools available to compressor designers making such a decision, without deviating significantly from their chosen model or architecture include techniques such as Amnesia or random deletion. *Contamination* is a generally-applicable modification to a broad family of compressors which presents the same ability to finely control run-time memory usage as Amne-

sia, but has been shown in a variety of cases to present improved compression effectiveness.

4.1 Future Work

Candidates worthy of investigation with *contamination* that are unexplored in this thesis are *infinite-depth* compressors such as PPM*, CTW*, and infinite-depth Deplump, which do not use an explicit context depth and typically require the use of compacted suffix trees (see subsection 2.1.2). It would be worthwhile to see if the same memory-effectiveness trade-off is seen between infinite-depth unrestricted compressors and infinite-depth *contaminating* compressors.[1]

Additionally, it might be interesting to investigate a hybrid Amnesia and *contamination* approach, which uses *contamination* normally, but under some heuristic condition resets all accumulated statistics as is done in Amnesia. Such a compressor might find a sweet spot where the accumulated statistics do not get too inaccurate from premature deletion in Amnesia, and not too inaccurate from the histogram collisions caused by prolonged use of *contamination*. A hybrid compressor may present better compression-effectiveness characteristics than simple *contaminating* compression, particularly in the case of power-law-based sequence models.

As a means of further improving power-law-based compressors under *contamination* and taking inspiration from PPM-DP's novel contributions: it may be worthwhile to add more context-distinguishing parameters for histogram weighting on top of the fanout-dependent and depth-dependent parameters. Specifically, a "histogram-corruption"-dependent parameter which changes as some function of the number unique contexts that map to a given histogram could weight the indicator function present in equation (2.7). Counting the number of unique contexts could be facilitated in constant space through the use of a fixed-size array, using a context hash as a unique identifier and employing similar saturating/clamping mechanisms as described in section 2.4. The optimisation of these additional parameters would follow the same approach used to optimize fanout and depth-dependent parameters.

Along the lines of incorporating context information in *contaminating* approaches, another method which could potentially improve the performance of power-law-based sequence models is a more intelligent collision design, or an intelligent method that decides when to collide histogram statistics of distinct contexts. Perhaps this method could incorporate existing histogram

[1] For an investigation of infinite-depth Deplump and Amnesia, see Bartlett, Pfau, and F. Wood (2010).

statistics when selecting where to place the statistics of a novel context. Such a method would require a small, per-histogram overhead to track which contexts map to the current histogram. Although an initial attempt may use a fixed-size array of context hashes, one could also use a *Bloom Filter* (B. H. Bloom (1970)) or another probabilistic data structure for set membership testing, whose accuracy degrades more gracefully a fixed-size array as the number of inserted elements increases.

Lastly, the excellent memory and compression effectiveness trade-off presented through the use of *contamination* in this thesis might be attractive in an ensemble setting such as PAQ from Mahoney (2005) as a means of further increasing ensemble size.[2]

[2] Due to the implementation of PAQ effectively being several hash-tables identifying different counts of different context occurrences (for example, N-grams of bytes, N-grams of text, N-grams gaps) where looked-up counts are combined in a neural network to arrive at conditional symbol probabilities, using *contamination* would mean reducing or eliminating collision detection mechanisms present for different context tables.

Appendix A

Computational Details

A.1 Efficiently Rescaling Narrow-Width Integers

Although integer division is typically a high-latency operation in most microarchitectures, divide-by-two-and-round-up can be efficiently implemented in terms of three efficient bitwise operations.

```
uint8_t rescale(uint8_t count):
    // Determine if count is odd.
    tmp = count & 0x1;
    // Divide by two, rounding down.
    count >>= 0x1;
    // Correct for odd numbers being rounded down.
    count += tmp;
    return count;
```

This procedure is not only amenable to vectorization (provided that all counts are stored contiguously), but also avoids short conditional jumps or predication-induced serialisation which would typically be required by a saturating scheme.

A.2 Incremental Computation in CTW

Following a proof given in Willems, Shtarkov, and Tjalkens (1995), under a binary alphabet for $|\mathcal{C}| = 0$ and $\mathrm{P}_{\mathrm{leaf}}(s \mid c) = \frac{1}{2}$, then if a symbol s is seen the following update is performed:

$$\mathrm{P}_{\text{leaf}}(s \mid c) \leftarrow \frac{\mathcal{C}[s] + \frac{1}{2}}{|\mathcal{C}| + 1} \cdot \mathrm{P}_{\text{leaf}}(s \mid c)$$

For a given update along a single path, the $\prod_{l \in \mathcal{A}_s}$ term from equation (2.1) only has a single changed value, which can be incrementally updated as well (provided appropriate weighting by $\frac{1}{2}$).

A.3 boost::hash_combine and the Choice of a Non-Cryptographic Hash Function

Boost's `hash_combine` is a state-less scheme for incrementally constructing a hash described in James (2006). Inspired initially by techniques presented in Hoad and Zobel (2003), the entire implementation can be written concisely as:

```
uint32_t hash_combine(uint32_t seed, uint32_t value) {
  return seed ^ (value + 0x9e3779b9 + (seed<<6) + (seed>>2));
}
```

Which has the appealing property of being exceptionally efficient, requiring just five bit-wise arithmetic operations, many of which can be conducted in parallel.

The magic number `0x9e3779b9` results from the binary expansion of the quotient formed from an irrational number: $\frac{2^{32}}{\phi} = \frac{2^{32}}{\frac{1+\sqrt{5}}{2}}$, and serves to avoid mapping $0 \rightarrow 0$.

However, `hash_combine` is a subpar hash function due to the fact that for distinct $\{a, b\}$, the probability of a collision, or that `hash_combine(seed, a)` equals `hash_combine(seed, b)` is much greater than the desirable $\frac{1}{2^p}$ which a good non-cryptographic, uniform hash would provide.

That being said, using a more standard (and slightly more computationally expensive) hash function which avoids these properties, such as the Fowler-Noll-Vo hash described in Fowler et al. (2019) does not yield empirical improvement in compression effectiveness.

```
uint32_t fnv_offset = // Some value
uint32_t fnv_prime = // Some value
uint32_t init_seed = fnv_offset;

uint32_t fnv_hash_combine(uint32_t seed, uint32_t value) {
```

```
    for (int i = 0; i < 32; i+=8) {
        char byte = (value >> i) & 0xFF;
        acc ^= byte;
        acc *= fnv_prime;
    }
    return acc;
}
```

The FNV hash function[1] is more computationally expensive as it works byte-wise and involves an integer multiplication for each byte, does not yield improved compression over the Calgary corpus, as shown in Figure A.1.

[1] The algorithm described above uses the FNV-1a variant which performs an exclusive-or before multiplying, which yields slightly better dispersion properties as described in Fowler et al. (2019, Section 2).

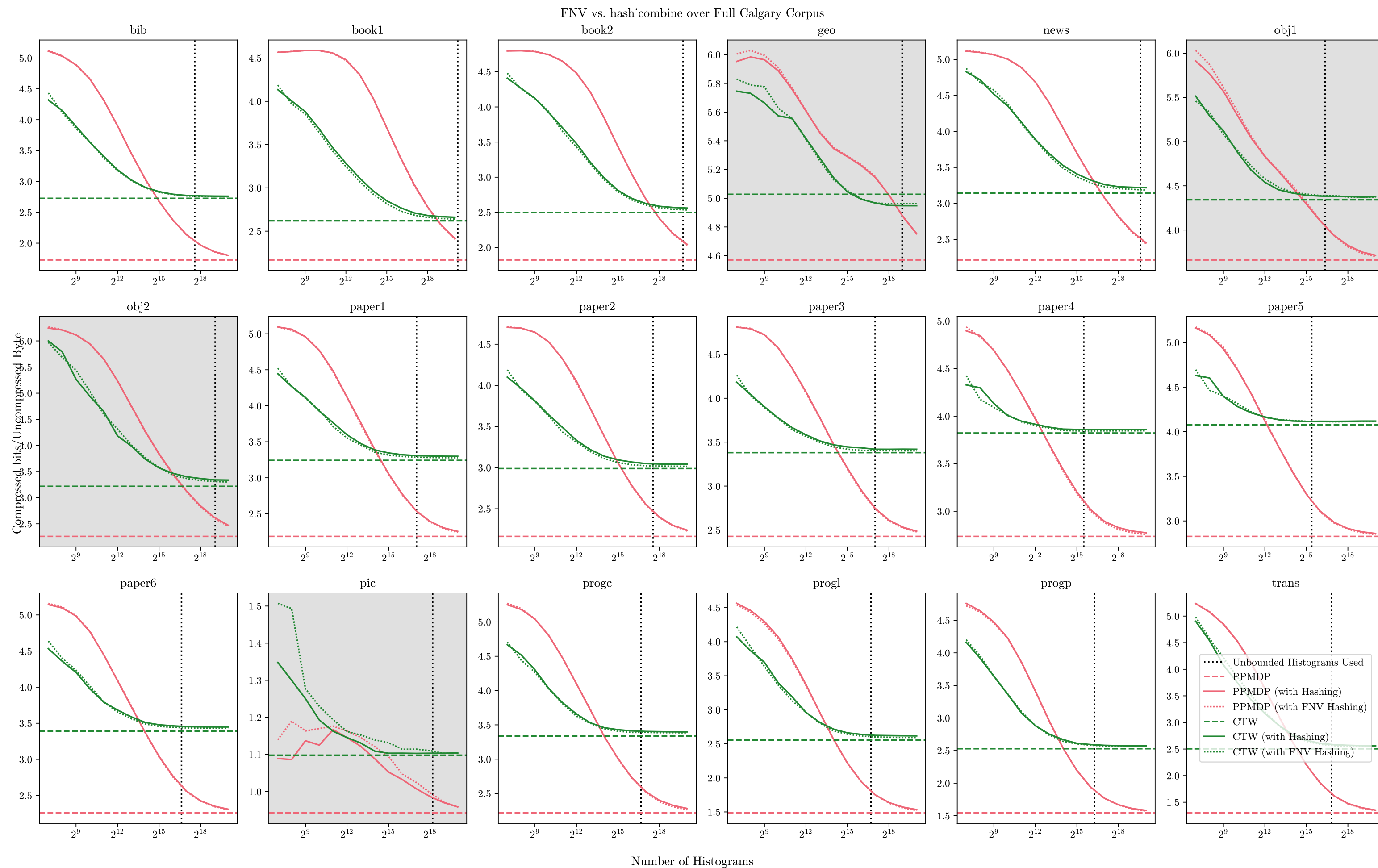


Figure A.1: Comparison of hash_combine and FNV as the hash function for *contaminating* variants of CTW and PPM-DP over the Calgary corpus. Notably, FNV and hash_combine curves nearly overlap, implying that the choice of hash function for a uniform hashing scheme does not drastically impact compression effectiveness.

Appendix B

Compression Results on Full Corpora

For brevity, only “vanilla” *contaminating* compression effectiveness is shown over both the Canterbury and Calgary corpora. All other variants discussed in chapter 3 are demonstrated over the Calgary corpus alone.

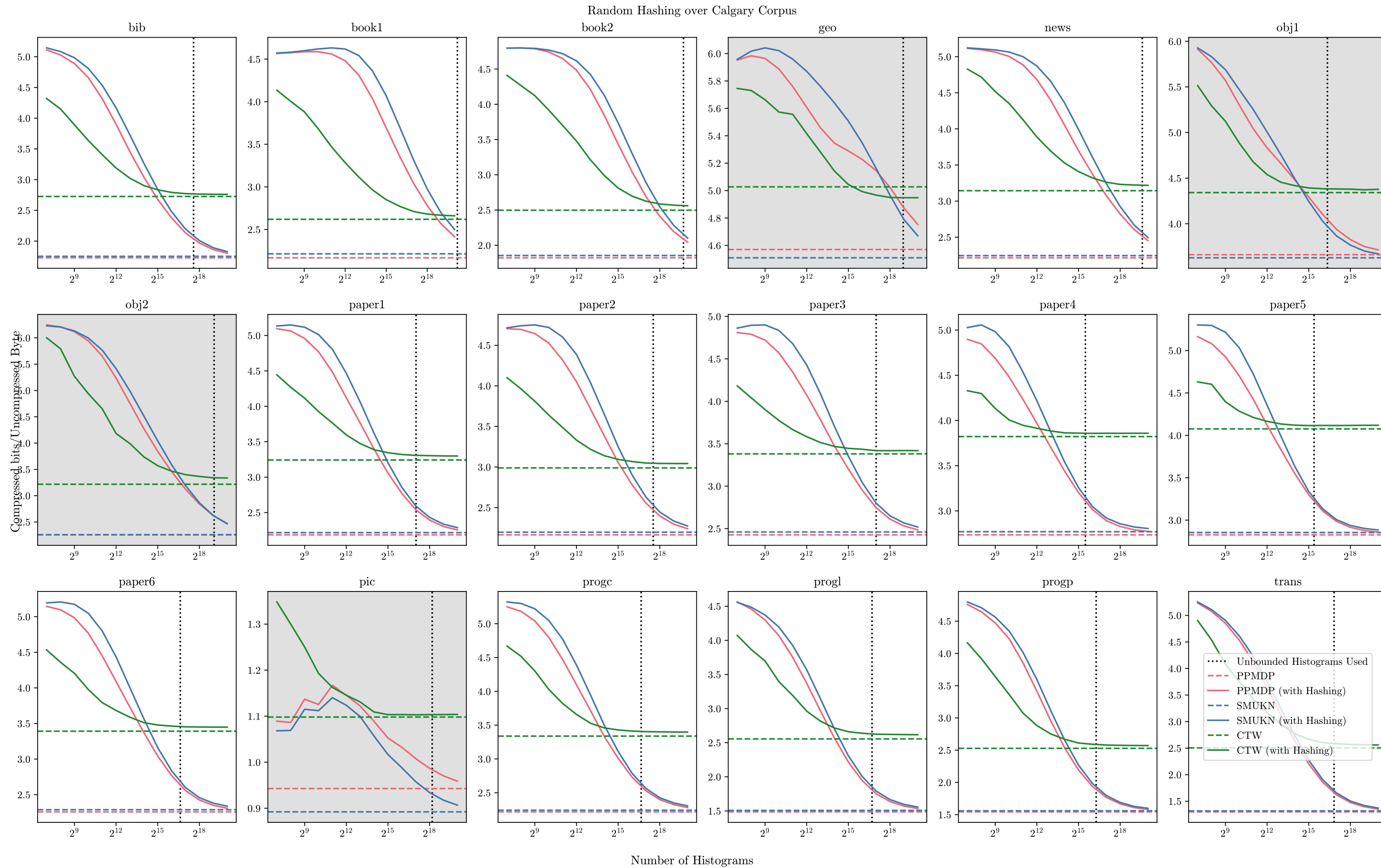


Figure B.1: Comparison of standard compression schemes all using a finite context depth of $D = 8$ with associated hash variants over the Calgary corpus. Dotted vertical line indicate memory usage and compression ratio of unbounded scheme. Grey figures denote that the test file is not human-readable text. Data points are sampled for every 2^p for integer $p \in [7, 21)$. SM1PF is omitted as it exhibits extremely similar compression effectiveness to SMUKN.

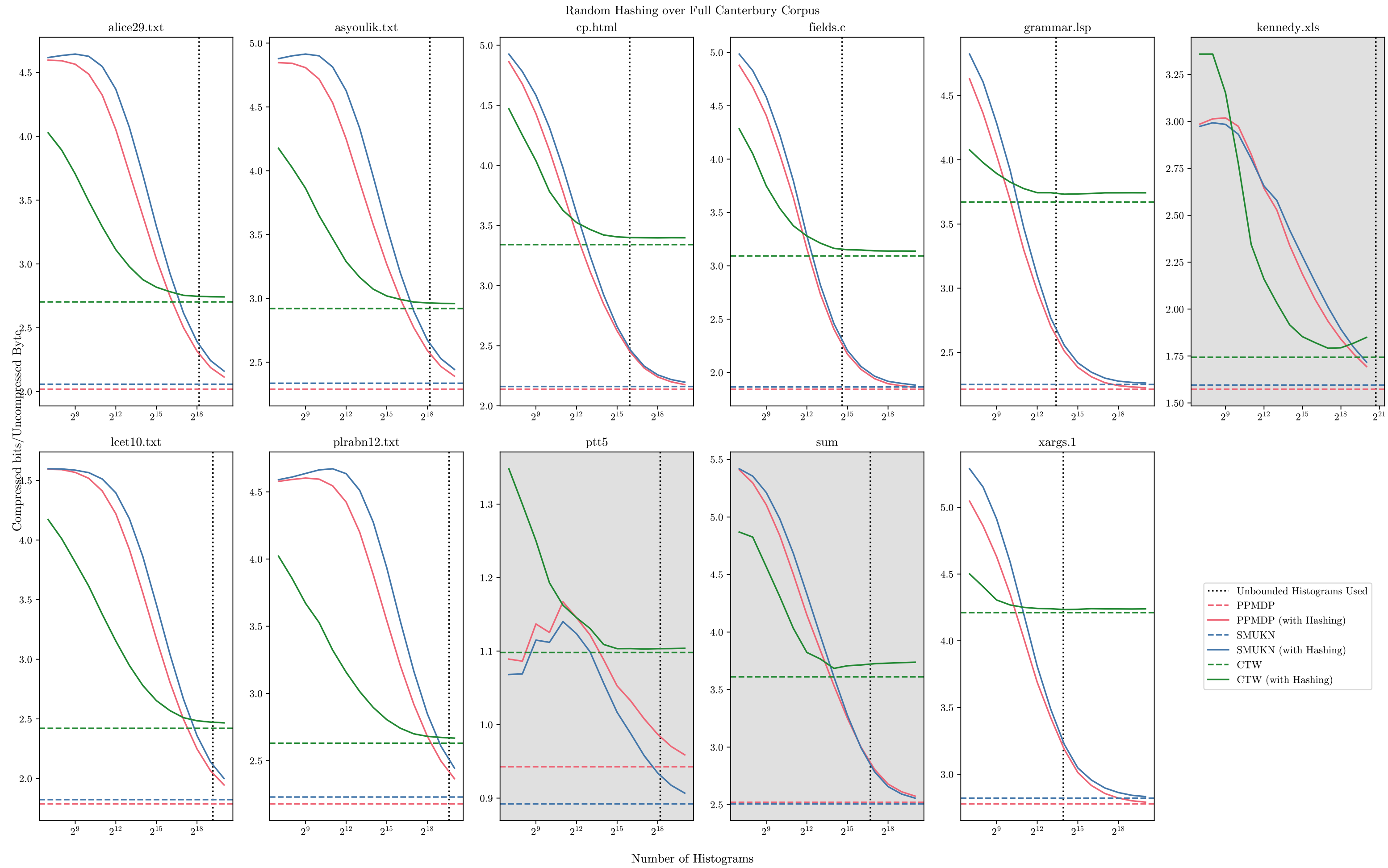


Figure B.2: Comparison of standard compression schemes with *contaminating* variant over the Canterbury corpus. Note that ptt5 is identical to pic from the Calgary corpus.

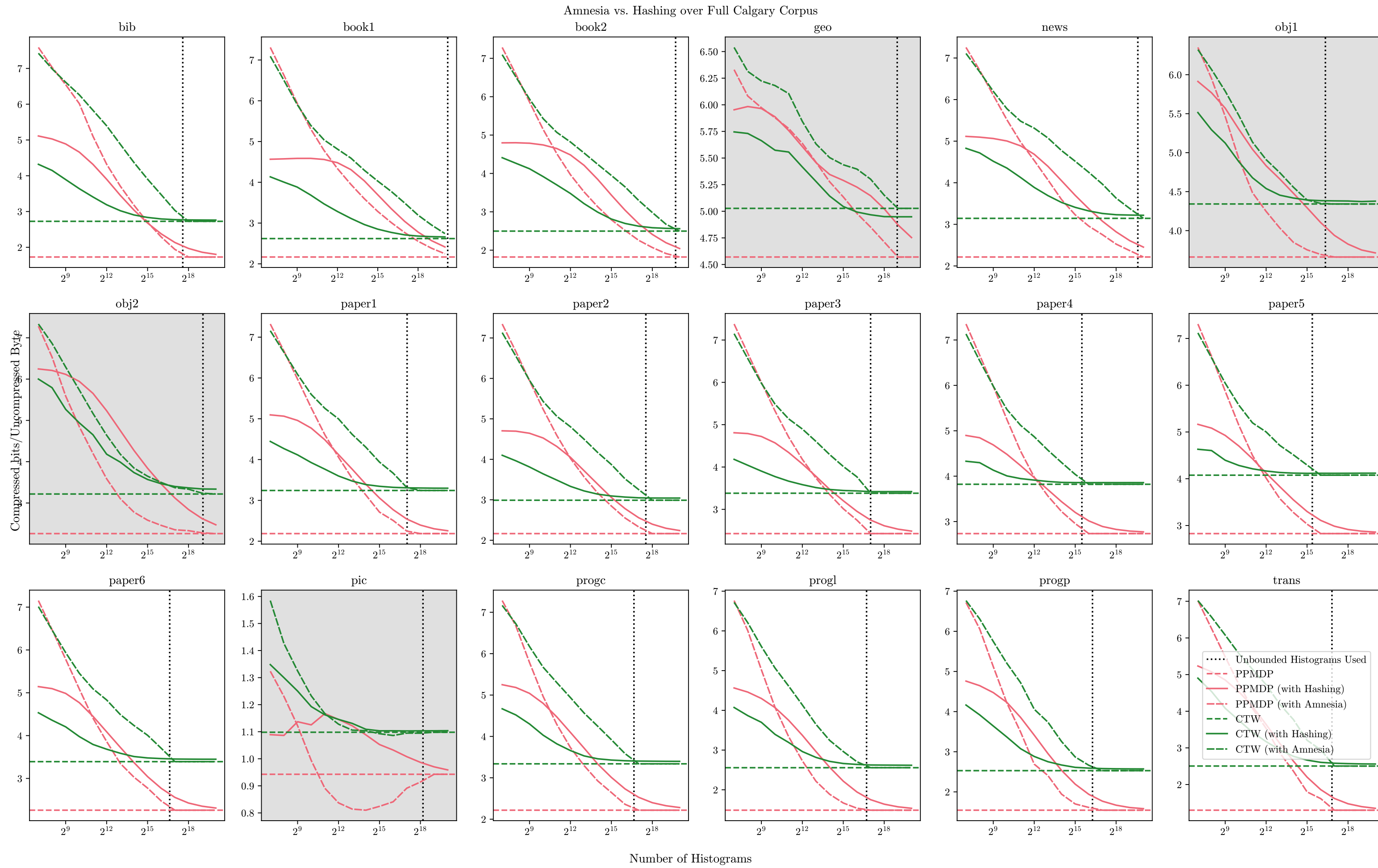


Figure B.3: Comparison of PPM-DP and CTW with *contaminating* and Amnesia variants over the Calgary corpus. For Amnesia, the independent variable represents the maximum number of Histograms the compressor is allowed to create, beyond which the backing Suffix Tree will be destroyed. SMUKN is omitted due to exhibiting similar behaviour as PPM-DP.

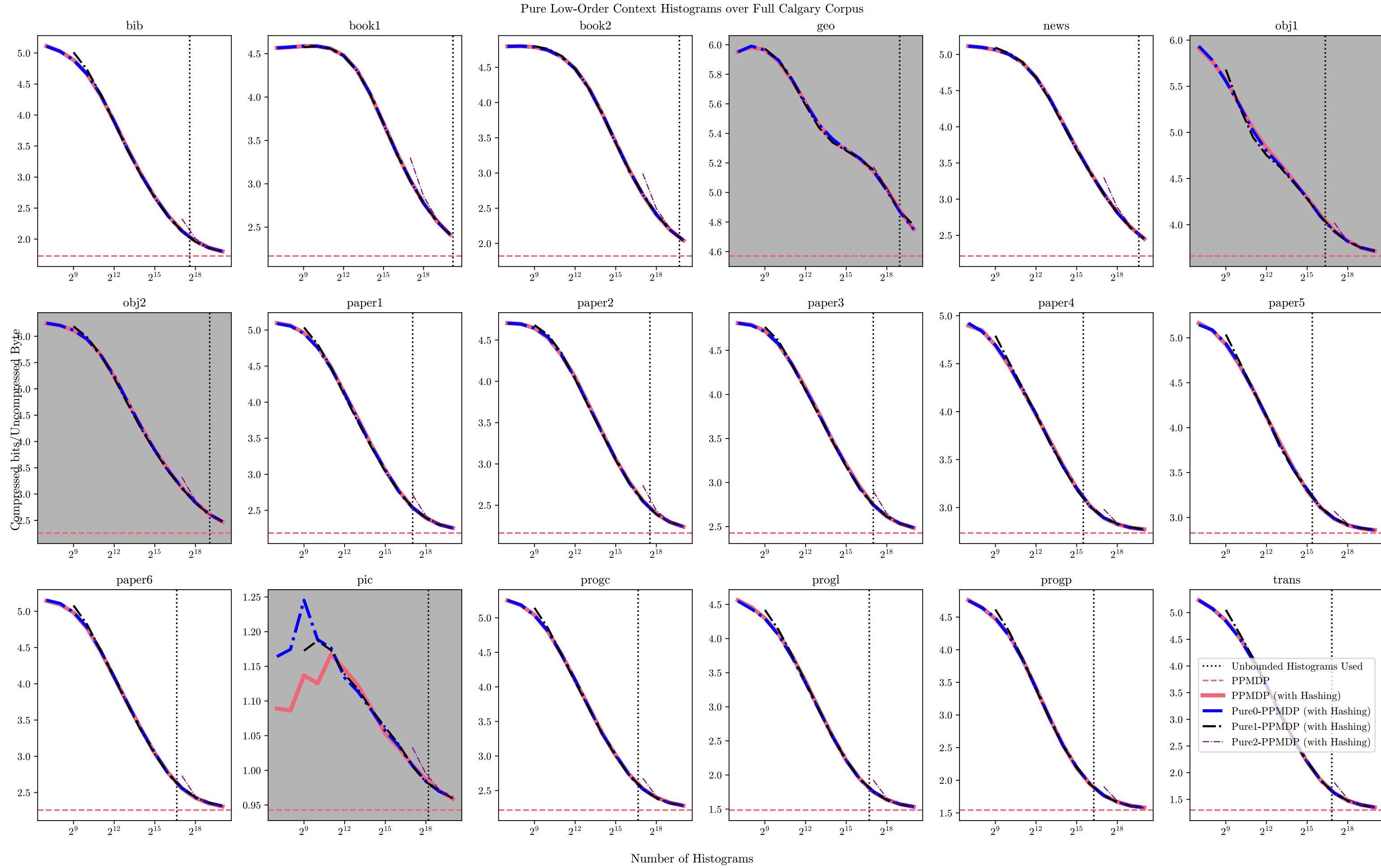


Figure B.4: Demonstration of HashPPM-DP with various amounts of reserved "pure" histograms. CTW and SMUKN are omitted due to exhibiting similar behaviour as PPM-DP in all cases. Decreasing line widths have been used to demonstrate how similarly all systems behave.

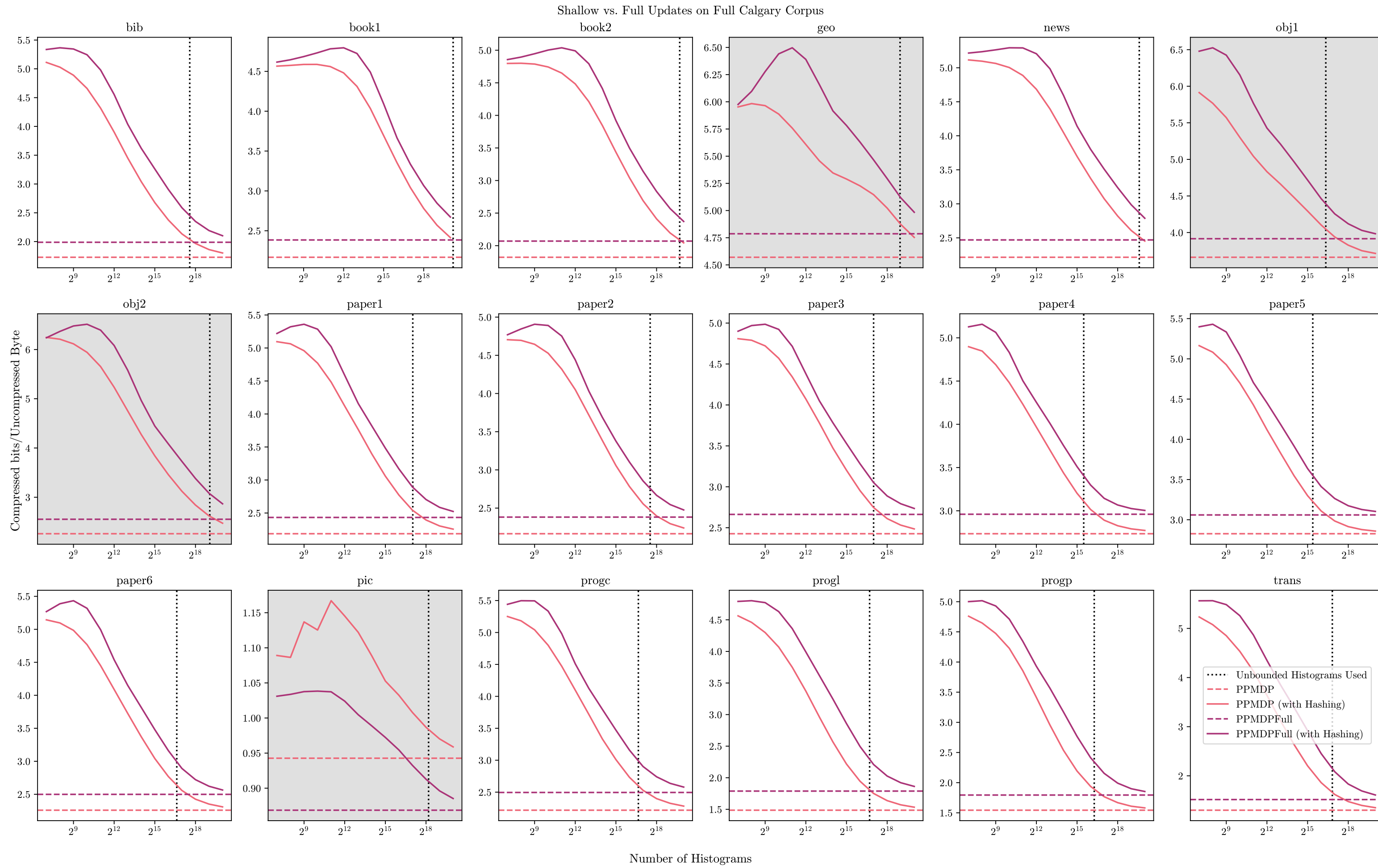


Figure B.5: Demonstration of HashPPM-DP and HashPPM-DPFull variants over the Calgary corpus. CTW omitted because it inherently uses full updates. SM variants omitted because they present near identical behaviour to PPM-DP.

Bibliography

Bartlett, Nicholas, David Pfau, and Frank Wood (Aug. 2010). "Forgetting Counts: Constant Memory Inference for a Dependent Hierarchical Pitman-Yor Process". In: *ICML 2010*, pp. 63–70.

Bell, Tim et al. (2000). *Description of the fax file.* URL: `https://corpus.canterbury.ac.nz/descriptions/cantrbry/fax.html`.

Bloom, Burton Howard (1970). "Space/Time Trade-Offs in Hash Coding with Allowable Errors". In: *Commun. ACM* 13.7, pp. 422–426. DOI: `10.1145/362686.362692`. URL: `https://doi.org/10.1145/362686.362692`.

Bloom, Charles (2010). *Hashes and Cache Tables.* URL: `http://cbloomrants.blogspot.com/2010/11/11-19-10-hashes-and-cache-tables.html`.

Chen, Stanley F. and Joshua Goodman (1998). *An Emperical Study of Smoothing Techniques for Language Modeling.* Tech. rep. 10-98. Harvard University.

Cleary, John G. and John J. Darragh (1984). *A Fast Compact Representation of Trees Using Hash Tables.*

Cleary, John G. and Ian H. Witten (1984). "Data Compression Using Adaptive Coding and Partial String Matching". In: *IEEE Transactions on Communications* 32.4, pp. 396–402. DOI: `10.1109/tcom.1984.1096090`. URL: `https://doi.org/10.1109%2Ftcom.1984.1096090`.

De La Briandais, Rene (1959). "File Searching Using Variable Length Keys". In: *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference.* IRE-AIEE-ACM '59 (Western). San Francisco, California: Association for Computing Machinery, pp. 295–298. DOI: `10.1145/1457838.1457895`. URL: `https://doi.org/10.1145/1457838.1457895`.

Fano, Robert M. (1949). *The Transmission of Information.* Tech. rep. 65. Massachusetts Institute of Technology.

Fowler, Glenn et al. (2019). *The FNV Non-Cryptographic Hash Algorithm.* Internet-Draft. Internet Engineering Task Force. URL: `https://datatracker.ietf.org/doc/draft-eastlake-fnv/17/`.

Franken, Erik and Marcel Peeters (2003). *CTW Overview: Storing the Context Tree in a Hash Table.* URL: `https://web.archive.org/web/`

20110823225156fw_/http://www.ele.tue.nl/ctw/overview/hashing.html.

Fredkin, Edward (1960). "Trie Memory". In: *Commun. ACM* 3.9, pp. 490–499. DOI: 10.1145/367390.367400. URL: https://doi.org/10.1145/367390.367400.

Gasthaus, Jan, Frank Wood, and Yee Whye Teh (Jan. 2010). "Lossless Compression Based on the Sequence Memoizer". In: *Proceedings of the 2010 Data Compression Conference*, pp. 337–345. DOI: 10.1109/DCC.2010.36.

Gosling, James et al. (2013). *Conversions and Promotions (From: The Java SE 7 Language Specification)*. URL: https://docs.oracle.com/javase/specs/jls/se7/html/jls-5.html.

Griffiths, Thomas et al. (2003). "Hierarchical topic models and the nested Chinese restaurant process". In: *Advances in neural information processing systems* 16.

Hancock, J. C. and J. L. Holsinger (1962). *Some Useful Coding Techniques for Binary Communication Systems*. Tech. rep. Purdue University.

Hoad, Timothy C and Justin Zobel (2003). "Methods for identifying versioned and plagiarized documents". In: *Journal of the American society for information science and technology* 54.3, pp. 203–215.

Howard, Paul G. and Jeffrey Scott Vitter (1992). "Practical Implementations of Arithmetic Coding". In: *Image and Text Compression*. Ed. by James A. Storer. Boston, MA: Springer US, pp. 85–112. DOI: 10.1007/978-1-4615-3596-6_4. URL: https://doi.org/10.1007/978-1-4615-3596-6_4.

Huffman, David A. (1952). "A Method for the Construction of Minimum-Redundancy Codes". In: *Proceedings of the IRE* 40.9, pp. 1098–1101. DOI: 10.1109/JRPROC.1952.273898.

Implicit Conversions (2022). URL: https://en.cppreference.com/w/c/language/conversion.

Iyigun, Mehmet and Seth Juarez (2015). *Memory Compression in Windows 10*.

James, Daniel (2006). *Boost container_hash Documentation: Combining Hash Values*. URL: https://www.boost.org/doc/libs/1_79_0/libs/container_hash/doc/html/hash.html#combine.

Kneser, Reinhard and Hermann Ney (1995). "Improved backing-off for m-gram language modeling". In: *1995 international conference on acoustics, speech, and signal processing*. Vol. 1. IEEE, pp. 181–184.

Lelewer, D.A. and D.S. Hirschberg (1991). "Streamlining context models for data compression". In: *Proceedings of the 1991 Data Compression Conference*. Pp. 313–322. DOI: 10.1109/DCC.1991.213349.

MacKay, David J.C. and Linda C. Bauman Peto (1995). “A hierarchical Dirichlet language model”. In: *Natural language engineering* 1.3, pp. 289–308.

Mahoney, Matthew Vincent (2005). *Adaptive weighing of context models for lossless data compression*. Tech. rep.

Martin, G. N. N. (1979). “Range Encoding: An Algorithm for Removing Redundance from a Digitised Message”. In: *Proceedings of the Video and Data Recording Conference*.

Marton, Yuval, Ning Wu, and Lisa Hellerstein (2005). “On Compression-Based Text Classification”. In: *Advances in Information Retrieval*. Ed. by David E. Losada and Juan M. Fernández-Luna. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 300–314.

McFadden, Andrew (1992). *Hacking Data Compression*. URL: `https://fadden.com/apple2/hdc/index.html`.

Pasco, Richard Clark (1976). “Source Coding Algorithms for Fast Data Compression”. PhD thesis. Stanford University.

Rein, S., C. Guhmann, and F.H.P. Fitzek (2006). “Low-complexity compression of short messages”. In: *Data Compression Conference (DCC’06)*, pp. 123–132. DOI: `10.1109/DCC.2006.45`.

Rissanen, Jorma (1976). “Generalized Kraft Inequality and Arithmetic Coding”. In: *IBM J. Res. Dev.*, pp. 198–203.

— (1986). “Complexity of strings in the class of Markov sources”. In: *IEEE Transactions on Information Theory* 32.4, pp. 526–532.

Rissanen, Jorma and Glen G. Langdon (1979). “Arithmetic Coding”. In: *IBM Journal of Research and Development* 23.2, pp. 149–162. DOI: `10.1147/rd.232.0149`.

— (1981). “Universal modeling and coding”. In: *IEEE Transactions on Information Theory* 27.1, pp. 12–23. DOI: `10.1109/tit.1981.1056282`. URL: `https://doi.org/10.1109%2Ftit.1981.1056282`.

Shannon, Claude E. (1948). “A mathematical theory of communication”. In: *The Bell System Technical Journal* 27.3, pp. 379–423. DOI: `10.1002/j.1538-7305.1948.tb01338.x`.

Stan, M.R. and W.P. Burleson (1995). “Bus-invert coding for low-power I/O”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 3.1, pp. 49–58. DOI: `10.1109/92.365453`.

Steinruecken, Christian (2014). “Lossless Data Compression”. PhD thesis. University of Cambridge.

Steinruecken, Christian, Zoubin Ghahramani, and David MacKay (2015). “Improving PPM with Dynamic Parameter Updates”. In: *2015 Data Compression Conference*, pp. 193–202. DOI: `10.1109/DCC.2015.77`.

Volf, Paulus Adrianus Jozef (2002). "Weighting Techniques in Data Compression: Theory and Algorithms". PhD thesis. Technische Universiteit Eindhoven.

Wagner, Bill et al. (2022). *Built-in numeric conversions (C# reference)*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/numeric-conversions.

Willems, Franz M.J., Yuri M. Shtarkov, and Tjalling J. Tjalkens (1995). "The context-tree weighting method: basic properties". In: *IEEE Transactions on Information Theory* 41.3, pp. 653–664. DOI: 10.1109/18.382012. URL: https://doi.org/10.1109%2F18.382012.

Witten, Ian H., Radford M. Neal, and John G. Cleary (June 1987). "Arithmetic Coding for Data Compression". In: *Commun. ACM* 30.6, pp. 520–540. DOI: 10.1145/214762.214771. URL: https://doi.org/10.1145/214762.214771.

Wood, Frank D. et al. (2009). "A stochastic memoizer for sequence data". In: *ICML*, pp. 1129–1136. URL: https://doi.org/10.1145/1553374.1553518.

Yang, Ke, Sian-Jheng Lin, and Honggang Hu (2018). "A Modified Version of Huffman Coding with Random Access Abilities". In: *Proceedings of 2018 IEEE 4th International Conference on Computer and Communications*, pp. 34–38.

Zipf, George Kingsley (1945). "The meaning-frequency relationship of words". In: *The Journal of general psychology* 33.2, pp. 251–256.