ARIEL UNIVERSITY

MASTER THESIS PROPOSAL

Weighted Adaptive Coding

Author:
Gross Yoav

Supervisor:
Prof. Shapira Dana

Department of Computer Science

August 1, 2022

Declaration of Authorship

I, Gross Yoav, hereby declare that this thesis proposal entitled, "Weighted Adaptive Coding" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

Ariel University

Abstract

Faculty of Natural Sciences
Department of Computer Science

Master

Weighted Adaptive Coding

by Gross Yoav

A new generic coding method is defined, extending the known static and dynamic variants and including them as special cases. This leads then to the formalization of a new adaptive coding method, which is shown to be always at least as good as the best dynamic variant known to date, and in particular, always better than static Huffman coding. We present empirical results that show improvements achieved by the proposed method, even when the encoded file includes the model description.

Contents

Chapter 1

Introduction

1.1 Introduction

Huffman coding [7] is one of the seminal techniques in data compression and is applied on a set of elements Σ into which a given input file $\mathcal{F}$ can be partitioned. We shall refer to Σ as an *alphabet* and to its elements as *characters*, but these terms should be understood in a broader sense, and the characters may consist of strings or words, as long as there is a well defined way to break $\mathcal{F}$ into a sequence of elements of Σ.

Huffman coding is known to be optimal in case the alphabet is known in advance, the set of codewords is fixed and each codeword consists of an integral number of bits. If one of these conditions is violated, optimality is not guaranteed.

In the *dynamic* variant of Huffman coding, also known as *adaptive*, the encoder and decoder maintain identical copies of the model; at each position, the model consists of the frequencies of the elements processed so far. After each processed element σ, the model is updated by incrementing the frequency of σ by 1, while the other frequencies remain the same. Specifically, Faller [4] and Gallager [6], propose a one-pass solution for dynamic Huffman coding. Knuth extends Gallager's work and also suggests that the frequencies may be decreased as well as increased [9], which enables the usage of a sliding window rather than relying on the full history. These independent adaptive Huffman coding methods are known as the FGK algorithm. Vitter [13] proposes an improved technique with additional properties and proves that the number of bits needed in order to encode a message of n characters by his variant, is bounded by the size of the compressed file resulting from the optimal two-pass static Huffman algorithm, *plus* n. In practice, Vitter's method produces often smaller files than static Huffman coding, but not always, and an example for which Vitter's dynamic Huffman coding produces a file that is larger can be found in [8].

An enhanced dynamic Huffman coding named *forward looking coding* [8] starts with the full frequencies, similar to the static variant, and then decreases them progressively. For this method, after each processed element σ, the model is altered by *decrementing* the frequency of σ by 1, while the other frequencies remain the same. Forward looking Huffman coding has been shown to be always better by at least $\Sigma - 1$ bits than static Huffman coding. We shall refer to the traditional dynamic Huffman coding as *backward looking*, because its model is based on what has already been seen in the past, unlike the forward looking variant that constructs the model based on what is still to come in the future. A hybrid method, exploiting both backward and forward approaches is proposed in [5], and has been shown to be always at least as good as the forward looking Huffman coding.

If the model is learned adaptively, as in the traditional backward looking codings, no description of the model is needed, since the model is updated by the encoder and the decoder in synchronization. However, in the other mentioned versions, the details of the chosen model on which the method relies, are needed for the decoding and should be adjoined to the compressed file, for example, as an header. For static coding this header may include approximate probabilities or just the set of codeword lengths. However, the forward looking as well as the hybrid variants require the exact frequencies of the elements. When the alphabet is small, the size of the necessary header might often be deemed negligible relative to the size of the input file. Larger alphabets, consisting, e.g., of all the words in a large textual database [10], may often be justified by the fact that the list of different words and their frequencies are needed anyway in an Information Retrieval system.

The contribution of this paper is as follows: we first define a new generic coding method which we call *weighted* coding, encompassing all mentioned variants (static, forward and backward) as special cases. Second, a new special case called *positional* is suggested, and shown to be always at least as good as the forward looking coding. Third, we present empirical results that show practical improvements of the proposed method, even when the encoded file includes the model description.

It is important to stress that all the methods can in fact be applied to every adaptive coding technique, in particular to arithmetic coding [14] or PPM [2]. This paper brings theoretical results only for Huffman coding, and empirical results for both Huffman and arithmetic coding.

This proposal is organized as follows. Section 2 introduces the weighted coding concept and shows how certain known compression techniques can be derived from it as special cases. Positional encoding is then proposed as a new variant, and the proof that it is at least as good as forward looking coding is given in Section 3. Section 4 presents empirical results for weighted codings with various parameters, showing its improvement in practice. We end this section with our work plan for further research.

Chapter 2

Weighted Coding

2.1 Definitions

Given is a file $T = T[1,n]$ of n characters over an alphabet Σ. We shall define a general weight $W(g,\sigma,\ell,u)$ based on four parameters, in which

- $g : [1,n] \longrightarrow \mathbb{R}^+$ is a non negative function defined on the integers that assigns a positive real number as a weight to each position $i \in [1,n]$ within T;
- $\sigma \in \Sigma$ is a character of the alphabet;
- ℓ and u are the boundaries of an interval, $1 \leq \ell \leq u \leq n$, serving to restrict the domain of the function g.

The value of the weight $W(g,\sigma,\ell,u)$ will be defined for each character $\sigma \in \Sigma$, as the sum of the values of the function g for all positions j in the range $[\ell,u]$ at which σ occurs, that is $T[j] = \sigma$. Formally

$$W(g,\sigma,\ell,u) \;=\; \sum_{\{\ell \leq j \leq u \;|\; T[j]=\sigma\}} g(j).$$

We are in particular interested in two kinds of weights, defined relatively to a current position i, one we call *Backward* looking and the other one *Forward* looking, or backward and forward weights for short. These are implemented by means of the interval $[\ell,u]$ used to restrict the considered range. The *backward weight* refers to the positions that have already been processed, i.e.,

$$W(g,\sigma,1,i-1) \;=\; \sum_{\{1 \leq j \leq i-1 \;|\; T[j]=\sigma\}} g(j),$$

whereas the *forward weight* corresponds to the positions yet to come,

$$W(g,\sigma,i,n) \;=\; \sum_{\{i \leq j \leq n \;|\; T[j]=\sigma\}} g(j).$$

The aim of this definition is to generalize different existing coding approaches into a consistent framework, so that they can be derived as special cases of weighted coding. This will then lead to the possibility of generating several new variants with improved performances, by varying the parameters to obtain hitherto unknown special cases.

STATIC CODING is the special case for which g is the constant function $\mathbb{1} \equiv g(i) = 1$ for all i, and the weight function, denoted by $W(\mathbb{1},\sigma,1,n)$ is constant for all indices.

The classical ADAPTIVE CODING is a special case of using a backward weight in which, as above, $g(i) = 1$ for all i, but unlike STATIC CODING, the weights are not constant and are rather recomputed for all indices $1 \leq i \leq n$ according to backward weights:

$$W(\mathbb{1}, \sigma, 1, i-1) = \sum_{\{1 \leq j \leq i-1 \mid T[j]=\sigma\}} 1 = \text{number of occurrences of } \sigma \text{ in } T[1, i-1].$$

FORWARD CODING is a special case of using a forward weight in which $g(i) = 1$ for all i. It is symmetrical to the classical adaptive coding, but computes its model according to suffixes rather than prefixes of the text T. That is,

$$W(\mathbb{1}, \sigma, i, n) = \sum_{\{i \leq j \leq n \mid T[j]=\sigma\}} 1 = \text{number of occurrences of } \sigma \text{ in } T[i, n].$$

The idea behind the extension below is the following. STATIC Huffman encodes a character σ by the same codeword $\mathcal{E}(\sigma)$, regardless of where in the text σ occurs. The choice of how many bits to allocate to $\mathcal{E}(\sigma)$ is therefore governed solely by the frequency of σ in T, and not by where in T the occurrences of σ can be found. In the ADAPTIVE approach, on the other hand, the set of frequencies in the entire file T are yet unknown after only a prefix of size $i-1$ has been processed, for $i \leq n$. Basing the encoding then on the currently known statistics is thus just an *estimate*, and the good performance of such an approach depends on whether or not the distribution of the characters derived from the processed prefix is similar to the distribution in the entire file. Backward adaptive methods take advantage of the fact that the damage of using wrong frequencies is limited, since the numbers and thus also the corresponding codewords are constantly updated, and the latter will ultimately achieve their optimal lengths. By reversing the process to consider the future rather than the past, the FORWARD coding again deals with correct frequencies, and not just with estimates.

However, once the psychological barrier forcing us to base our encoding models on frequencies or their estimates has been broken, it might be justified to deviate from the common practice and try a *greedy* approach for a more convenient definition of the model. In particular, characters that are close to the current position might be assigned a higher priority than those farther away. The rationale of such an assignment is that the close by characters are those that we are about to encode, so we concentrate on how to reduce the lengths of their codewords, even at the price of having to lengthen the codewords of more distant characters in the text, since, anyway, the encoding of those will be reconsidered by the adaptive process once we get closer to them.

The assignment of differing priorities or *weights* can be materialized by using a decreasing function g instead of a constant one. The simplest option would be a linear decrease, which leads to the following definition.

POSITIONAL CODING, first defined in this paper, is a special case of a forward weight, with $\mathcal{L} \equiv g(i) = n - i + 1$ for $1 \leq i \leq n$, where $n = |T|$. We shall use the notation $p_\sigma(i)$ to denote $W(\mathcal{L}, \sigma, i, n)$.

Note that the idea of giving increased attention to closer rather than to more distant elements is not new to data compression. A similar choice appears when choosing a sliding window of limited size in Ziv-Lempel coding [12], and when the

i	1	2	3	4	5	6	7	8	9	10
T	c	c	a	b	b	b	c	a	a	a
$\mathcal{L}(i)$	10	9	8	7	6	5	4	3	2	1
$p_{\mathtt{a}}(i)$	14	14	14	6	6	6	6	6	3	1
$p_{\mathtt{b}}(i)$	18	18	18	18	11	5	0	0	0	0
$p_{\mathtt{c}}(i)$	23	13	4	4	4	4	4	0	0	0

TABLE 2.1: *Positional Coding example for* $T = \mathtt{ccabbbcaaa}$.

accumulated frequencies are periodically rescaled in adaptive (backward) Huffman or arithmetic coding, as suggested in [11].

As we shall see in the experimental section below, the intuition of assigning higher weights to closer elements pays off and indeed yields improvements in the compression performance. This leads naturally to pushing the idea even further. The extreme case would be an exponentially decreasing function g, e.g., $\mathcal{E} \equiv g(i) = 2^{n-i}$. In this case, the weight of the following character to be processed will always be the largest, since even when the suffix of the text is of the form $\mathtt{abbb}\cdots\mathtt{b}$, we get that $W(\mathcal{E}, \mathtt{a}, i, n) = 2^{n-i} > \sum_{j=i+1}^{n} 2^{n-j} = W(\mathcal{E}, \mathtt{b}, i, n)$. It follows that the codeword assigned by Huffman's algorithm at position i will be of length 1 bit. Therefore, using this exponential function $\mathcal{E}$ for g, the text will be encoded by exactly n bits, one bit per character, which means that the bulk of the information is encoded in the header.

This encoding of the header is, however, very costly. The weight of each of the characters may be of the order of 2^n, requiring $\theta(n)$ bits for its encoding, so the header may be of size $O(|\Sigma|n)$. Moreover, the update algorithms will be very time consuming, having to deal with numbers of unbounded precision. The challenge is therefore to find reasonable functions g, which yield a good tradeoff between encoding the text and the header, and we aim at minimizing the sum of their sizes.

2.2 Detailed comparative example

To clarify these definition, we present the text $T = \mathtt{c\ c\ a\ b\ b\ b\ c\ a\ a\ a}$ as a small running example and compare its different encodings.

2.2.1 Positional coding

Recall that we use $p_\sigma(i)$ to denote $W(\mathcal{L}, \sigma, i, n)$. The details for the Positional encoding are presented in Table 2.1. The function $\mathcal{L}$, given on the second line, enumerates the indices in reverse order starting at $n = 10$ down to 1. At the first position $i = 1$, the values of $p_{\mathtt{a}}(1)$, $p_{\mathtt{b}}(1)$ and $p_{\mathtt{c}}(1)$ are 14, 18 and 23, respectively, as shown in the first column of the last three rows. In a left to right scan, the values $p_\sigma(i)$ only change at indices i for which $T[i-1] = \sigma$. Light gray therefore refers to the non-changed values (starting, in a left to right scan, just after an occurrence of σ and ending at the rightmost position where σ occurs in T).

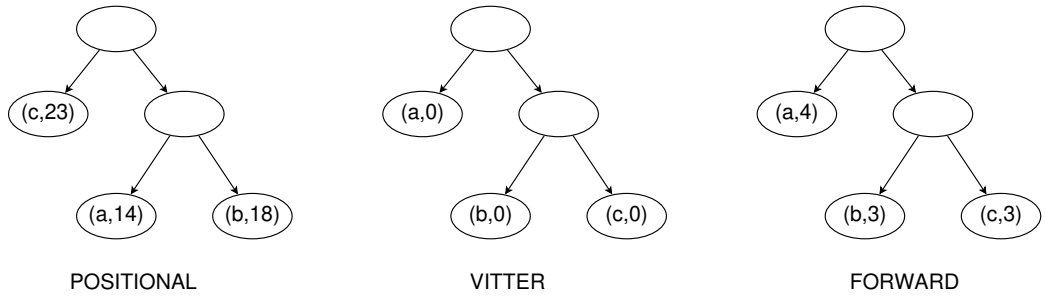

FIGURE 2.1: *Initial trees for* $T = \texttt{ccabbbcaaa}$.

The Huffman tree is initialized with the weights of position 1, as shown in the left part of Figure 2.1. The first c is encoded by 0, and its weight is then decremented by $\mathcal{L}(1) = 10$ from 23 to 13. The tree gets updated, now having leaves with weights 14, 18 and 13 for a, b and c, respectively, as given in column 2 of Table 2.1. The second c is therefore encoded by the two bits 10. The weight of c is decremented by $\mathcal{L}(2) = 9$ from 13 to 4, and the following character a is encoded by the two bits 11. The weights are then updated to 6, 18 and 4, as shown in column 4 of Table 2.1. At this stage, the character b has become the one with the shortest codeword, and the two following bs are encoded each by 0, updating the weight of b first to 11 and then to 5, so that the encoding for the last b becomes 11. After the last b is processed, it is removed from the tree as its frequency has become 0, resulting in a tree containing only c and a. When the last c is processed, the codeword 0 is output, the leaf for c is removed from the tree, and the tree remains with a single node corresponding to a. Since the decoder also discovers that the alphabet of the remaining suffix of the file contains only a single character, which must be a, with weight $p_a(8) = \sum_{i=8}^{10} \mathcal{L}(i) = 6$, the number of its repetitions can be calculated, thus no additional bits need to be transferred.

2.2.2 Vitter's backward coding

The model in Vitter's algorithm need not be transmitted to the decoder as it is learnt incrementally while processing the encoded file. We assume that the exact alphabet is known to both encoder and decoder, and do not use the special *Not-Yet-Transmitted* leaf suggested by Vitter for dealing with newly encountered symbols. The initial tree for Vitter's algorithm is the middle one in Figure 2.1, starting with all frequencies equal to 0.

The correctness of Vitter's algorithm relies on the *sibling property* [6], which states that a tree is a Huffman tree if and only if its nodes can be listed by nonincreasing weight so that each node is adjacent to its sibling. We shall use the convention to place the nodes in a Huffman tree in such a way that this list can be obtained by a bottom-up, left to right scan of the nodes.

The codeword for c is 11, and its frequency is incremented by 1, resulting in a shorter codeword, 0, for the following c, by swapping the leaves of a and c. The next character a is encoded by 11, and its frequency is updated, but the tree remains unchanged. The following two characters are b and b, both of which are encoded by 10. The frequency of b gets updated first to 1, then to 2, resulting in a swap with the leaf for a in order to retain the sibling property. For the next and last b the codeword 11 is output, its frequency is increased to 3, and its leaf is swapped with the leaf corresponding to c. The following character c results in the output of the codeword

11 and an update of its frequency to 3. The codewords for each of the last three as is then 10, incrementing weight of a from 1 to 2, 3, and 4, where the tree is changed only at the last step, only after the last a has already been encoded.

2.2.3 Forward coding

The FORWARD coding algorithm basically works in the opposite way, starting with the final tree of the dynamic variant, and ending with the empty tree. The tree is initialized with weights $(W(\mathbb{1}, \mathtt{a}, 1, n), W(\mathbb{1}, \mathtt{b}, 1, n), W(\mathbb{1}, \mathtt{c}, 1, n)) = (4, 3, 3)$, like for static Huffman encoding, as shown in the right part of Figure 2.1. The first c is encoded by 11, and the frequency of c is decremented to 2, resulting in an interchange of b and c. The second c is therefore encoded by 10, and its frequency is updated to 1. The character a is then encoded by 0 and its weight is decremented to 3. The following three bs are encoded by 11 and the frequency for b is repeatedly decremented to 0 and its leaf is finally removed from the tree. The tree remains with 2 leaves for a and c, and the following character c is encoded by 0. The last three as need not be encoded as for positional coding.

	Weights			c	c	a	b	b	b	c	a	a	a
	a	b	c										
POSITIONAL	14	18	23	0	10	11	0	0	11	0	–	–	–
VITTER	–	–	–	11	0	11	10	10	11	11	10	10	10
FORWARD	4	3	3	11	10	0	11	11	11	0	–	–	–

FIGURE 2.2: *The encoding of* $T =$ ccabbbcaaa *using the three adaptive techniques.*

Figure 2 summarizes the different encodings of this example. The first three columns of the table display the initial weights of the symbols; these should be encoded and prepended to the compressed file. The rest of the table shows the binary output sequences produced by the different approaches. Although all the reported outcomes of our experimental results include the appropriate header for each method (this header is empty for Vitter's algorithm), we do not include any precise binary encoding in this small example; it is generally of secondary importance relative to the size of real life input files, but it might distort the outcome of the comparison on a small artificial example as this one.

The net number of bits required to encode T for this example by the three alternatives is 10, 19 and 12 for POSITIONAL, VITTER and FORWARD, respectively. Note that the first a is encoded by a single bit by FORWARD and by two bits by POSITIONAL, illustrating that although there are overall savings in space, the individual codewords assigned by FORWARD may be locally shorter than the corresponding ones of POSITIONAL.

Chapter 3

Analysis

3.1 Analysis

This section provides a proof showing that POSITIONAL coding is at least as good as FORWARD, which in turn has been proven to be better than STATIC Huffman coding by at least $|\Sigma| - 1$ bits.

Lemma 3.1.1 *Given an index t, $1 \leq t < n$, then encoding $T[1,t]$ using* FORWARD *for the first t characters, that is, with weights $W(\mathbb{1}, \sigma, 1, t)$ restricted to the prefix of length t, followed by using* FORWARD *for the last $n-t$ characters of T with weights $W(\mathbb{1}, \sigma, t+1, n)$ restricted to the suffix of length $n-t$, is at least as good as using* FORWARD *for the entire message, i.e., using weights $W(\mathbb{1}, \sigma, 1, n)$.*

Proof: The difference between the encodings of applying FORWARD to the entire message, versus applying it to the first and second parts separately, is the encoding of the prefix of T up to position t. Splitting the encoding into two parts considers, while encoding the first part, only the number of occurrences of any character σ within the first t characters, rather than in the entire text of n characters. The latter also takes into account the occurrences of σ in the suffix of T of size $n-t$, which are not relevant for encoding just the prefix of size t. The resulting encoding using the split into two parts can therefore not be worse.

We actually need a generalization of Lemma 3.1.1 to work for weighted adaptive coding and not only for the special case of FORWARD. The generalized case is given in the following Lemma.

Lemma 3.1.2 *Given an index t, $1 \leq t < n$, a non negative function $g : [1,n] \longrightarrow \mathbb{R}^+$, and two parameters $0 \leq k_1 \leq k_2$, then encoding $T[1,t]$ with weights*

$$W(\mathbb{1}, \sigma, 1, t) + k_1 \cdot W(g, \sigma, t+1, n),$$

(taking into account also occurrences of σ in the suffix $T[t+1,n]$ of T), is at least as good as encoding $T[1,t]$ with weights

$$W(\mathbb{1}, \sigma, 1, t) + k_2 \cdot W(g, \sigma, t+1, n).$$

Proof: Note that we concentrate on encoding the prefix of T up to position t. The first summand $W(\mathbb{1}, \sigma, 1, t)$ computes the number of occurrences of any character σ within the first t characters, and the optimality of Huffman's algorithm is obtained for this true distribution. The second summand, however, $k_i \cdot W(g, \sigma, t+1, n)$, considers also the number of occurrences of σ in the suffix of T of size $n-t$ multiplied by some constant k_i, so that the resulting weights deviate from the true distribution. This deviation increases with larger weights in the second summand. Therefore, the

resulting encoding for the prefix of T up to position t using k_1 cannot be worse than the corresponding encoding using k_2.
Lemma 3.1.2 generalizes Lemma 3.1.1, which corresponds to the particular choice of the parameters $k_1 = 0$, $k_2 = 1$, and function $g = \mathbb{1}$.

Theorem 3.1.3 *For a given file T of length n, the average codeword length of* POSITIONAL *is at least as good as the average codeword length of* FORWARD *coding.*

Proof: We construct a sequence of functions $\mathcal{G} = \{g_j\}_{j=1}^{n}$ as follows, the first one g_1 being the function corresponding to FORWARD and the last one g_n to POSITIONAL. The function g_1 is thus the constant function $\mathbb{1} \equiv g_1(i) = 1$. For $j \geq 2$, we define g_j recursively by:

$$g_j(i) = \begin{cases} j & i \leq (n-j+1) \\ g_{j-1}(i) & i > (n-j+1), \end{cases}$$

so g_j is constant up to $n-j+1$ and then decreases linearly, see Table 2 for an illustration.

	1	2	3	4	5	6	7	8	9	10
T	c	c	a	b	b	b	c	a	a	a
$g_1(i)$	1	1	1	1	1	1	1	1	1	1
$g_2(i)$	2	2	2	2	2	2	2	2	2	1
$g_3(i)$	3	3	3	3	3	3	3	3	2	1
$g_4(i)$	4	4	4	4	4	4	4	3	2	1
					...					
$\mathcal{L}(i)$	10	9	8	7	6	5	4	3	2	1

TABLE 2: *Positional Coding example.*

We show that the encoding based on g_{j+1} is at least as good as the encoding based on g_j, for all j, so that ultimately, POSITIONAL is at least as good as FORWARD.

Consider first the suffix $T[n-j+1, n]$ of T. The weights $W(g_j, \sigma, n-j+1, n)$ and $W(g_{j+1}, \sigma, n-j+1, n)$ for all characters $\sigma \in \Sigma$ are identical for both functions, as visualized in Figure 3, so the encoding of the suffix $T[n-j+1, n]$ will be the same for g_j and for g_{j+1}.

	1	2	3	...	$n-j-1$	$n-j$	$n-j+1$	...	$n-2$	$n-1$	n
g_j	j	j	j		j	j	j	$j-1$	3	2	1
g_{j+1}	$j+1$	$j+1$	$j+1$		$j+1$	$j+1$	j	$j-1$	3	2	1

FIGURE 3.1: *Schematic view of function g_j and g_{j+1}.*

For encoding the first $n-j$ positions, we consider the weights for the entire interval $[1, n]$ based on functions g_j and g_{j+1}, as illustrated in Figure 3.1, and rewrite the weights as follows.

$$W(g_j, \sigma, 1, n) = j \cdot \Big(W(\mathbb{1}, \sigma, 1, n-j) + \frac{1}{j} \cdot W(g_j, \sigma, n-j+1, n) \Big),$$

and

$$W(g_{j+1}, \sigma, 1, n) = (j+1) \cdot \Big(W(\mathbb{1}, \sigma, 1, n-j) + \frac{1}{j+1} \cdot W(g_j, \sigma, n-j+1, n) \Big),$$

because the weights W for g_j and g_{j+1} are the same for the suffix starting at $n-j+1$. Since $\frac{1}{j+1} < \frac{1}{j}$, we can apply Lemma 3.1.2 and get that encoding $T[1, n-j]$ with weights $W(\mathbb{1}, \sigma, 1, n-j) + \frac{1}{j+1} \cdot W(g_j, \sigma, n-j+1, n)$ is at least as good as encoding $T[1, n-j]$ with weights $W(\mathbb{1}, \sigma, 1, n-j) + \frac{1}{j} \cdot W(g_j, \sigma, n-j+1, n)$. The multiplication by j and $j+1$, respectively, changes the absolute weights, but preserves the *relative* weights for all $\sigma \in \Sigma$, and thus, the corresponding encodings.

Summing up the encodings for the prefix $T[1, n-j]$ of T and suffix $T[n-j+1, n]$ of T concludes the proof.

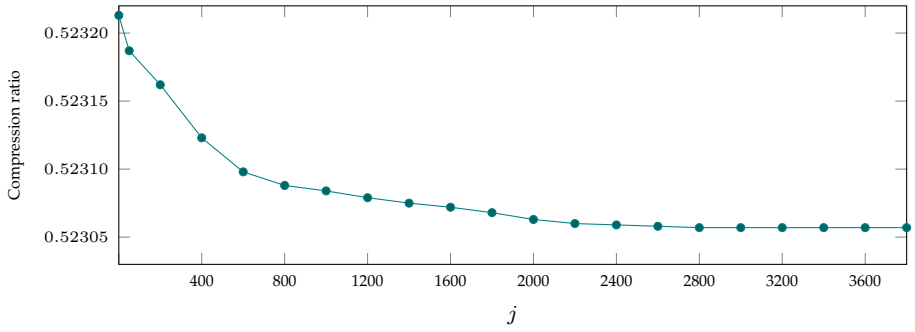

FIGURE 3.2: *Compression ratio for* g_j.

To illustrate the behavior of the family of functions $\mathcal{G}$, Figure 3.2 shows the relative size of the compressed file for selected values of j on our test file to be described below; as expected, the resulting curve is decreasing.

Chapter 4

Experimental Results

4.1 Experimental Results

To get empirical evidence how the weighted encoding behaves in practice, we considered files of different languages, sizes and nature, and obtained similar results. We thus report here only the performance on the King James version of the English Bible (in which the text has been stripped of all punctuation signs), as example for typical behavior. In order to handle the arithmetic of the huge numbers that are necessary for the coding, we used the GNU Multiple Precision Arithmetic Library[1]. Indeed, on some of our tests, the involved numbers used up to 800 bits. We applied the weighted compression with two different families of functions, using both Huffman and arithmetic coding.

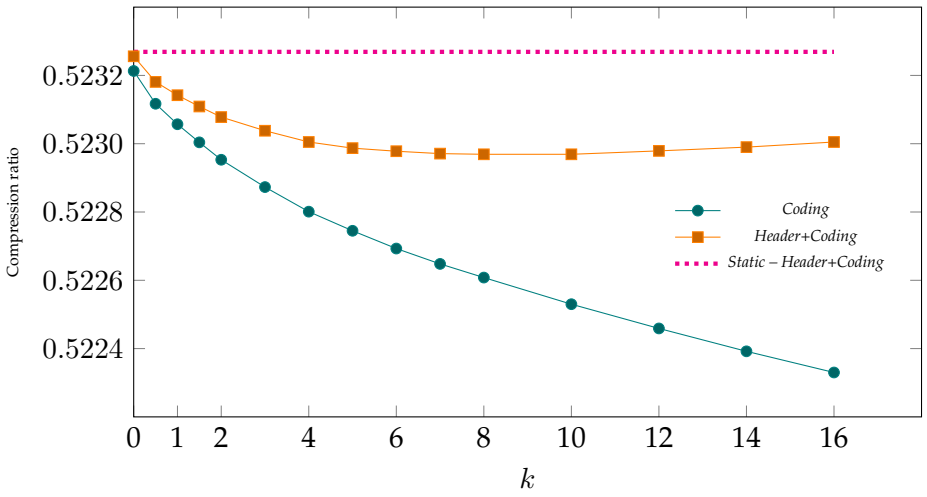

FIGURE 4.1: Compression efficiency of weighted Huffman encoding for $g(i) = (n - i + 1)^k$.

We first considered weighted coding corresponding to functions of the form $g(i) = (n - i + 1)^k$. Figures 4.1 and 4.2 present the compression ratio, defined as the size of the compressed divided by the size of the original file, for integer values of k ranging from 0 to 16, as well as for $k = 0.5$ and $k = 1.5$ using both Huffman and arithmetic coding, respectively. In particular, FORWARD is the special case $k = 0$, and POSITIONAL encoding corresponds to $k = 1$. The lower plot of each graph gives the net encoding while the upper ones include also the necessary header. As can be seen, the compression efficiency improves as k increases in both variants, until about $k = 8$ for Huffman, and $k = 10$ for arithmetic coding, where the combined

[1] https://gmplib.org/

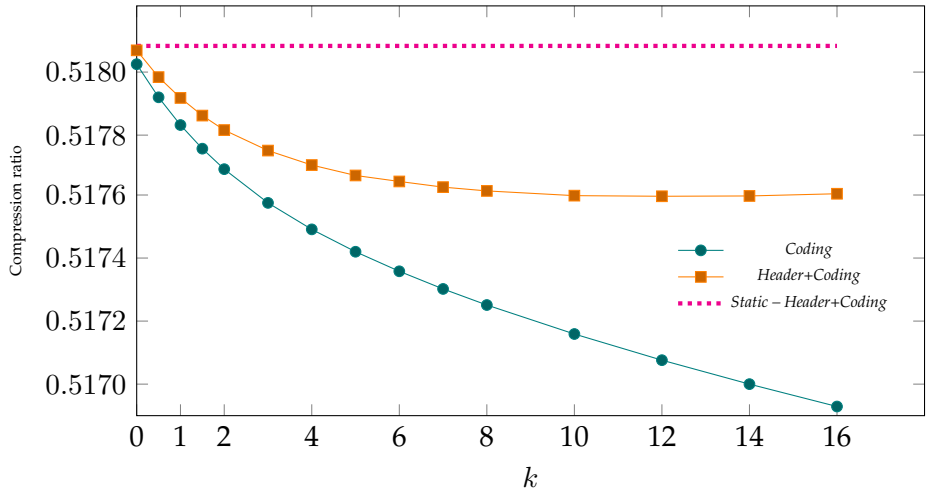

FIGURE 4.2: Compression efficiency of weighted arithmetic encoding for $g(i) = (n - i + 1)^k$.

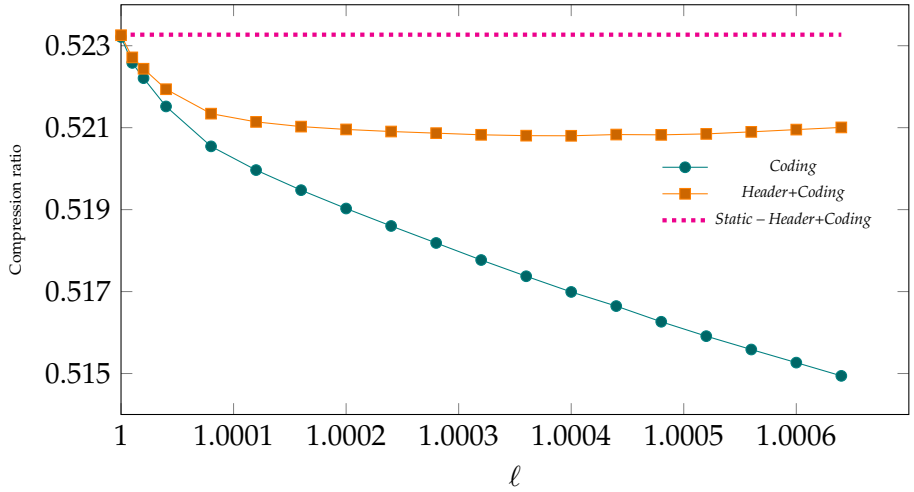

FIGURE 4.3: Compression efficiency of weighted Huffman encoding for $g(i) = \ell^{n-i}$.

(file + header) sizes start to increase, being still better than the size for FORWARD. The compression ratio for static Huffman and static arithmetic coding are given for comparison.

The family of functions $g(i) = (n - i + 1)^k$ considered in the first set of experiments does not retain a constant ratio between consecutive positions i, which yields a bias towards higher values of i. For example, referring to our running example of positional Huffman, position 1 is given the weight $p(1) = 10$, but the sum of weights from this position on is $\sum_{i=1}^{10}(10 - i + 1) = 55$, so the relative weight for $i = 1$ is $10/55 = 0.18$; on the other hand, $p(8) = 3$ yielding a relative weight for position 8 of $3/(3 + 2 + 1) = 0.5$. In our following experiment, we thus considered a more balanced family of functions, $g(i) = \ell^{n-i}$, where ℓ is a real number slightly larger than 1, which retains a ratio of ℓ between consecutive indices. Figures 4.3 and 4.4 follow the same format as Figures 4.1 and 4.2. This time FORWARD HUFFMAN corresponds to $\ell = 1$, and again an improvement is achieved. On this family of functions, the

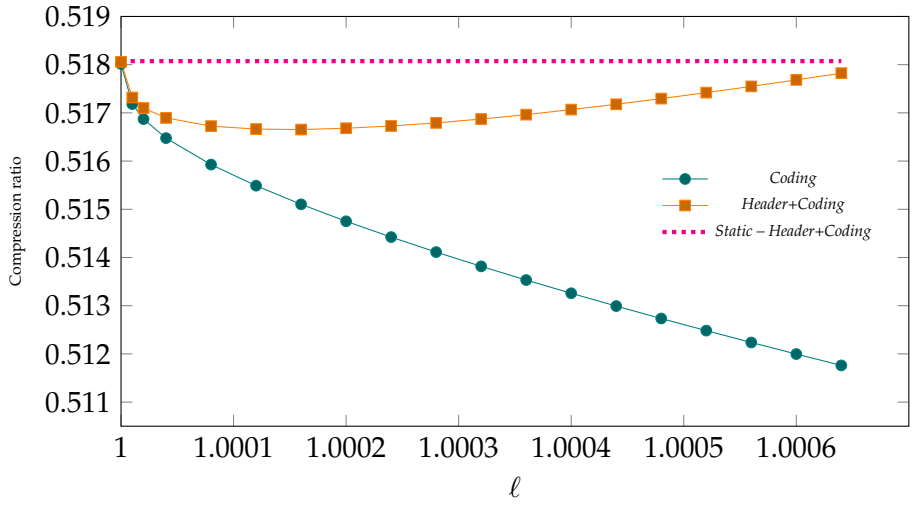

FIGURE 4.4: Compression efficiency of weighted arithmetic encoding for $g(i) = \ell^{n-i}$.

optimal combined size of file plus header is obtained for $\ell = 1.0004$ for the Huffman implementation and and for $\ell = 1.00016$ for the arithmetic coding implementation.

We see that for both families of weight functions, there is an evident improvement in the compression performance, though only a slight one on the given test file. The significance of our contribution is indeed not the derivation of a ground breaking new compression method, but rather the theoretical and empirical evidence that simple approaches which have been believed to be optimal for years, might at times be improved.

4.2 Future Work

In the weighted forward methods, the size of the prelude is a significant factor in the quality of compression. This part describes the total weight of each character, depending on the selected function. The binary representation of each of the values may require even hundreds and thousands of bits. Therefore, a *Uniquely Decipherable* variable length code may be preferable for encoding the numerical values, possibly concentrating on a prefix-free codes. For our empirical results the prelude was based on the sorted ASCII table and the Elias's C_δ [3] code was used to encode the function values of the symbols. C_δ code is a universal encoding method for integers ≥ 1 so that each value n is encoded by about $\log n + \log\log n$ bits. In future research we would like to investigate other alternatives, described as follows, for encoding huge numerical values in more economical space.

- The information is arranged as in the previous method, but encoded by C_ϵ instead of C_δ. That is, in C_ϵ the length of the binary representation is encoded by C_δ followed by the binary representation without the leading 1-bit. This method shortens the coding of the binary representation length, only for very large values, such as those required in some of the experiments. In this method, encoding the number n requires about $\log n + \log\log n + \log\log\log n$ bits.

- In this method we wish to reduce the values to be coded, by sorting the weights in ascending order, and encoding only the differences between successive weights. Since the characters are not arranged in a predetermined order, the ASCII code must be explicitly passed along with each weight.

- Another direction of investigation would be to reduce the values to be transmitted. Moreover, we suggest to include additional information in the prelude, so that the encoding of the lengths of the binary representations of the numerical values becomes redundant, and can, therefore, be omitted. That is, in case Elias encoding is used to encoded a numerical value x, the prefix of the codeword of x is the number of bits in its binary representation. If the decoder knows the interval in which x occurs in, some of the bits in this prefix can be removed. Using this idea, we suggest an encoding that is based on a binary tree rather than a universal code. In this algorithm, as a preliminary step, we will encode the minimum and maximum weights using a universal code, such as C_δ. Then, insert the rest of the weights into a binary search tree, and scan it by preorder traversal. In this method, the value passed is the difference from the known minimum corresponding to the subtree, as in the previous method. In addition, the range of possible values for each weight is bounded by the maximum, and it decreases as the depth of the tree decreases.

Entropy compression techniques mainly depend on two parameters: the distribution and dispersion of the involved symbols. Another research direction will focus on the second parameter.

A main characteristic of the proposed weighted encoding is giving higher relative weights to closer indices that results in shorter encodings. We would like to push this idea even further, by locally classifying characters together. To take advantage of this feature, we want to examine the effect of the Burrows–Wheeler transform [1] on the encoding. This reversible transformation is expected to group identical characters together.

We would also like to propose an enhanced variant of the proposed weighted encoding as follows. Let g denote the weight function, and i denote the index on which the encoder stands on. If there exists some character σ that satisfies $W(\sigma, g, i, n) < g(i)$, we know for sure that $T[i] \neq \sigma$. Therefore, at each step we can filter these characters out from the model, thereby increasing the relative weight of the remaining characters. As example, for the extreme function $g(x) = 2^x$, at each index i only a single character will fulfill the equation $W(\sigma, g, i, n) \geq g(i)$. So the pure encoding, ignoring the huge prelude, will be empty, as compared to a single bit per character using the proposed method without filtering.

As a last goal of my thesis I intend to extend Theorem 3.1.3 to other functions. The generalization will prove that the average codeword length corresponding to any decreasing monotonic function is at least as good as the average codeword length of FORWARD coding. In particular, the extended claim includes the decreasing functions $g(i) = (n - i + 1)^k$ and $g(i) = l^{n-i}$. The results of the corresponding experiments indeed fit the above assumption.

Bibliography

[1] Michael Burrows and David J Wheeler. "A block-sorting lossless data compression algorithm". In: (1994).

[2] John Cleary and Ian Witten. "Data Compression Using Adaptive Coding and Partial String Matching". In: *IEEE Transactions on Communications* 32.4 (1984), pp. 396–402.

[3] Peter Elias. "Universal codeword sets and representations of the integers". In: *IEEE Trans. Information Theory* 21.2 (1975), pp. 194–203.

[4] Newton Faller. "An adaptive system for data compression". In: *Record of the 7-th Asilomar Conference on Circuits, Systems and Computers*. 1973, pp. 593–597.

[5] Aharon Fruchtman, Shmuel T. Klein, and Dana Shapira. "Bidirectional Adaptive Compression". In: *Proceedings of the Prague Stringology Conference 2019*. 2019, pp. 92–101.

[6] Robert Gallager. "Variations on a theme by Huffman". In: *IEEE Transactions on Information Theory* 24.6 (1978), pp. 668–674.

[7] David Huffman. "A method for the construction of minimum redundancy codes". In: *Proc. of the IRE* 40 (1952), pp. 1098–1101.

[8] Shmuel T. Klein, Shoham Saadia, and Dana Shapira. "Forward Looking Huffman Coding". In: *The 14th Computer Science Symposium in Russia, CSR, Novosibirsk, Russia, July 1-5*. 2019.

[9] Donald E. Knuth. "Dynamic Huffman coding". In: *Journal of Algorithms* 6.2 (1985), pp. 163–180.

[10] Alistair Moffat. "Word-based Text Compression". In: *Softw., Pract. Exper.* 19.2 (1989), pp. 185–198.

[11] Mark Nelson and Jean-Loup Gailly. *The Data Compression Book, 2nd Edition*. M & T Books, 1996.

[12] James A. Storer and Thomas G. Szymanski. "Data compression via textural substitution". In: *J. ACM* 29.4 (1982), pp. 928–951.

[13] Jeffrey S. Vitter. "Design and analysis of dynamic Huffman codes". In: *J. ACM* 34.4 (1987), pp. 825–845.

[14] Ian H Witten, Radford M Neal, and John G Cleary. "Arithmetic coding for data compression". In: *Communications of the ACM* 30.6 (1987), pp. 520–540.