PPM based Forward Compression

Din Avrunin[1], Shmuel T. Klein[2], Rachel Mustakis Avrunin[1], and Dana Shapira[1]

[1] Dept. of Computer Science, Ariel University, Ariel 40700, Israel
{din.avrunin,rachush2}@gmail.com, shapird@g.ariel.ac.il

[2] Dept. of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel
tomi@cs.biu.ac.il

Abstract. A new variant of dynamic encoding, named Forward Looking, has been recently proposed, that provably always performs better than general static encoding, such as Huffman or arithmetic coding. This paper suggests to integrate Forward Looking with the well known adaptive *Prediction by Partial Matching* algorithm. This combination, that attempts to predict the following character based on the context that has already occurred in past, but uses its knowledge of the exact frequencies in the future, is shown to enhance the prediction capability, and therefore improve the compression efficiency.

Keywords: Lossless Compression, Arithmetic Coding, PPM

1 Introduction

Data compression techniques are often partitioned into static and adaptive algorithms. Alternatively, they can be classified by whether being statistical or dictionary based methods. In this research we combine the well known adaptive PPM [1] algorithm with a recently introduced compression paradigm named Forward Looking, which is based on statistical coding such as Huffman [5] and arithmetic coding.

Huffman coding is one of the foundations of data compression algorithms, used in both lossless and lossy techniques, and is well-known for its optimality, while still being simple. Given is a text $T = x_1 \cdots x_n$ over some alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_m\}$ with a corresponding probability distribution $P = \{p_1, \ldots, p_m\}$, such that the probability of σ_i in T is p_i. The problem is to assign binary codewords of lengths ℓ_i bits to the characters σ_i, such that the set of codewords is *uniquely decipherable*, and such that the expected codeword length $\sum_{i=1}^{m} p_i \ell_i$, given in bits, is minimized. If the restriction that all the codeword lengths ℓ_i have to be integers is removed, then an optimal assignment of lengths, would be the *information content* $\ell_i = -\log p_i$, and the average codeword length would then be $H = -\sum_{i=1}^{n} p_i \log p_i$, called the *entropy*. This can be reached by applying *arithmetic coding* [16].

Static compressors encode a character using the same codeword throughout the text. Static codes can be of fixed lengths, such as the *American Standard Code for Information Interchange* ASCII code, or variable length codes, such as Huffman [5], Elias [2] and Fibonacci coding [9]. Though many compression methods are based on the use of variable length codes, there are also certain methods in which the lengths of the codewords are more restricted, which can be useful for fast decoding and compressed searches [12, 8].

Adaptive compressors allow the model to be constructed dynamically by both the encoder and the decoder during the course of the transmission, and have been shown

to incur a smaller coding overhead than explicit transmission of the model's statistics. A family of adaptive dictionary methods were introduced by Ziv and Lempel, in particular, LZ77 [17] and LZ78 [18]. Storer and Szymanski [11] proposed a practical variant of the LZ77 compressor, named LZSS, which is based on the principle of finding redundant strings or patterns and replacing them by pointers to a common copy. LZW [15] is a practical method for LZ78, where the dictionary is initialized by the single characters of the alphabet, and then is updated dynamically by adding newly encountered substrings that have not been seen previously in the parsing of the underlying text.

Prediction by Partial Matching (PPM) is yet another adaptive coding method, which was proposed by Cleary et al. [1] as a combination of statistical coding with a Markov model, and is known as one of the best lossless compression algorithms to date. Their experimental results show that English texts can be coded using 2.2 bits per character on average with no prior knowledge of the source [10]. Despite its space efficiency, the technique suffers from slow processing time and thus the use of this method is not as common as expected.

Forward looking adaptive compression, is a variant of dynamic statistical encoding, and has been proven to be better than static Huffman compression [6], which is known to be optimal under certain constraints. This is achieved by reversing the direction for the references of the encoded elements to those forming the model of the encoding, from pointing backwards into the past to looking forward into the future. The lower bound shown for FORWARD LOOKING on the size of its compressed file is smaller than the lower bound of other dynamic statistical coding methods. An extension of the FORWARD LOOKING algorithm to *bidirectional* adaptive compression was proposed in [4], taking both past and future into account, and its net encoding is provably at least as good as the future-only based variant.

The new proposed integrated algorithm presented herein is at least as good as the original PPM compression. Our paper is organized as follows. Section 2 and Section 3 recall the details of Forward looking and PPM, respectively. Section 4 combines the two algorithms, and presents a new implementation of the PPM algorithm that is based on FORWARD.

2 Forward looking dynamic Huffman coding

A model consists of two main components, the alphabet elements and their corresponding statistics. For fixed length codes the model is quite primitive and utilizes a uniform distribution over the given alphabet. The model for static codes is more advanced and assigns a fixed distribution to the alphabet symbols throughout the encoding of the entire file, whereas the model for adaptive codes may update both the alphabet and the distribution of the involved elements while processing the input file.

The standard way for updating the model of adaptive codes is according to what has already been seen in the input file processed so far. The distribution of the following element to be encoded at some current location in the file is determined according to the distribution of the elements that have occurred up to that position. In case the exact number of occurrences of each element in the entire file are known, e.g., when this information can be obtained by a preprocessing scan of the file, a

different adaptive approach, named FORWARD LOOKING, can be applied [6]. In this approach, the dynamic model gets adjusted according to the information of what is still to come, i.e., it looks into the *future*, as opposed to what is done by traditional dynamic methods, which base their current model on what has already been seen in the *past*.

Utilizing the knowledge of what is still to come has also been proposed in [7] for performing streaming pattern matching in LZSS [11] compressed files, where the locations of the references to reoccurring strings have been moved and their direction has been reversed to point forwards, rather than letting the Ziv-Lempel type (offset, length) ordered pairs to point backwards, as in the original encoding.

Traditional encoding algorithms concentrate on the present element that is being processed and increments its probability, implying a decrease in its information content. However, consequently, the information content of certain other elements may increase, and therefore the entropy of the entire text. The FORWARD LOOKING paradigm provides a "social" approach that takes into account all elements, rather than only the one that is currently getting processed. In this method the probability of the processed element is *decreased*, even at the price of the corresponding information content becoming larger. However, this operation may reduce the overall entropy, yielding better space savings.

The Forward looking coding can be applied to any statistical coding such as Huffman or arithmetic coding. It has been proven to be more effective than the original static Huffman coding and thus theoretically more efficient than the dynamic Huffman algorithm of Vitter [14] since it relies on known statistics instead of learning them "on the fly". The proof can be easily extended to show the advantage of Forward over static arithmetic coding, which surprisingly implies a provably better than zero-order entropy encoding (see for example [3]).

The general Forward algorithm initialized its model identically to the standard static version. That is, the Huffman variant of Forward starts with the same Huffman tree as the static one, where each leaf refers to a certain element of the alphabet, and contains its frequency in the entire text. If the Forward arithmetic variant is utilized, each interval is partitioned into sub-intervals whose sizes are proportional to the probabilities in the distribution of the alphabet, as in the static arithmetic coding. Once the model is initialized as explained above, the general step of FORWARD, consists of the two following statements:

1. encode the next element based on the current model;
2. update the model by decrementing the frequency of the current element and adjusting the distribution accordingly.

	0	1	2	3	4	5	6	7
T	l	o	s	s	l	e	s	s

As example, consider the text $T =$ `lossless`. The alphabet Σ is $\{\mathtt{e}, \mathtt{l}, \mathtt{o}, \mathtt{s}\}$ with frequencies $\{1, 2, 1, 4\}$, respectively. The initial distribution is

$$P = \{p(\mathtt{e}) = \frac{1}{8}, \quad p(\mathtt{l}) = \frac{1}{4}, \quad p(\mathtt{o}) = \frac{1}{8}, \quad p(\mathtt{s}) = \frac{1}{2}\},$$

and the character `l` at position 0 is encoded with probability $\frac{1}{4}$, having information content of 2 bits. The frequency for `l` is then decremented by 1 reflecting the fact that only a single `l` remains in the text, and the updated distribution is

$$P = \{p(\texttt{e}) = \frac{1}{7}, \quad p(\texttt{l}) = \frac{1}{7}, \quad p(\texttt{o}) = \frac{1}{7}, \quad p(\texttt{s}) = \frac{4}{7}\}.$$

The symbol `o` is then encoded with probability $\frac{1}{7}$, and eliminated from the alphabet, as it is the last occurrence of `o`. The distribution gets updated to

$$P = \{p(\texttt{e}) = \frac{1}{6}, \quad p(\texttt{l}) = \frac{1}{6}, \quad p(\texttt{s}) = \frac{2}{3}\}.$$

The character `s` is then encoded with probability $\frac{2}{3}$, and the distribution is updated to

$$P = \{p(\texttt{e}) = \frac{1}{5}, \quad p(\texttt{l}) = \frac{1}{5}, \quad p(\texttt{s}) = \frac{3}{5}\},$$

and so on. Note that when only a single symbol remains in the text to be processed, no additional encoding is needed, as the decoder also realizes this case. Therefore, the last two symbols `s` are not encoded. Figure 1 presents the information content of each character of T as compared to static and adaptive arithmetic coding.

Arithmetic coding represents T by a real number a in the range $0 \leq a < 1$. An interval $[\ell, h)$ is initialized by $[0, 1)$, and is partitioned into sub-intervals relatively to the probability distribution of the characters. The procedure continues with the sub-interval selected according to the current processed symbol of T. That is, the interval gradually gets narrowed as more characters from T are processed. The real number a is then chosen from the final interval, preferably being a number with economical representation size.

Continuing with our running example, the initial interval $[0, 1)$ is partitioned according to the static distribution

$$P = \{p(\texttt{e}) = \frac{1}{8}, \quad p(\texttt{l}) = \frac{1}{4}, \quad p(\texttt{o}) = \frac{1}{8}, \quad p(\texttt{s}) = \frac{1}{2}\},$$

and the sub-interval $[\frac{1}{8}, \frac{3}{8})$ referring to `l`, the first character of T, is selected. The range is of length $\frac{1}{4}$, corresponding to information content of 2 bits. The range $[\frac{7}{32}, \frac{1}{4})$ referring to `o` is then selected with length $\frac{1}{32}$, having information content of five bits, and so on. The number of bits added to the encoded file by each processed character is exactly its information content, for a total of the entropy. The figures are given in the first column of Figure 1.

Unlike the static arithmetic coding in which the partition of $[0, 1)$ into sub intervals is fixed throughout the process, the dynamic variant updates this partition adaptively, according to the probability distribution of the alphabet within the prefix of the text that has already been processed. The frequency of the currently processed character is incremented and the relative sizes of all the intervals in the partition are adjusted accordingly.

The initial partition of the interval $[0, 1)$ for our running example is to 4 equal sub intervals of size $\frac{1}{4}$, suiting the uniform distribution, and `l` is encoded by 2 bits. The frequency of `l` is then incremented by 1, and the distribution is updated to

$$P = \{p(\texttt{e}) = \frac{1}{5}, \quad p(\texttt{l}) = \frac{2}{5}, \quad p(\texttt{o}) = \frac{1}{5}, \quad p(\texttt{s}) = \frac{1}{5}\},$$

and o is encoded with probability $\frac{1}{5}$. The frequency of o is then incremented, and the distribution is updated to

$$P = \{p(\mathtt{e}) = \frac{1}{6}, \quad p(\mathtt{l}) = \frac{1}{3}, \quad p(\mathtt{o}) = \frac{1}{3}, \quad p(\mathtt{s}) = \frac{1}{6}\}.$$

The character s is encoded with probability $\frac{1}{6}$, and so on. Adaptive arithmetic coding applied on our running example is presented in the third column of Figure 1. The entropy of the entire text T, also computed as the information content of the final interval of STATIC and ADAPTIVE arithmetic coding is given in the last row of Figure 1.

$T[i]$	Static		adaptive		Forward	
	p_i	$-\log(p_i)$	p_i	$-\log(p_i)$	p_i	$-\log(p_i)$
l	$\frac{1}{4}$	2.000	$\frac{1}{4}$	2.000	$\frac{1}{4}$	2.000
o	$\frac{1}{8}$	3.000	$\frac{1}{5}$	2.322	$\frac{1}{7}$	2.807
s	$\frac{1}{2}$	1.000	$\frac{1}{6}$	2.585	$\frac{2}{3}$	0.585
s	$\frac{1}{2}$	1.000	$\frac{2}{7}$	1.807	$\frac{3}{5}$	0.737
l	$\frac{1}{4}$	2.000	$\frac{1}{4}$	2.000	$\frac{1}{4}$	2.000
e	$\frac{1}{8}$	3.000	$\frac{1}{9}$	3.170	$\frac{1}{3}$	1.585
s	$\frac{1}{2}$	1.000	$\frac{3}{10}$	1.737	1.0	0.000
s	$\frac{1}{2}$	1.000	$\frac{4}{11}$	1.459	1.0	0.000
Total		14.000		17.080		9.714

Figure 1. The information content of Forward Looking as compared to Static on $T =$ lossless.

3 PPM compression

Prediction by Partial Matching (PPM) is an adaptive compression algorithm which is based on statistical encoding. The main idea is to encode each symbol in the sequence, in the framework of its context. It is based on the known inequality that for any random variables X and Y, the conditional entropy of X and Y, $H(X|Y)$, is less or equal than the entropy of X, $H(X)$, i.e.,

$$H(X|Y) \leq H(X).$$

Given a text $T = t_1 t_2 \cdots t_n$ of length n, the encoding of the following element t_i based on the context of its k previous symbols $t_{i-k} \cdots t_{i-1}$, $k \geq 1$, may increase the probability of the appearance of t_i, and thus its information content, which allows the encoding of t_i to use fewer bits.

The parameter k_{max} is defined as the maximum allowed context size, that is, the number of examined previous characters that are used in order to predict and encode the current one. k_{max} is also named the *order* of PPM, and is typically less or equal to 8, otherwise, the space for storing the model becomes too large.

The process for encoding each symbol t_i, $1 \leq i \leq n$, is initialized by assigning $k = k_{max}$. If no prediction to t_i can be made based on its k previous symbols $t_{i-k} \cdots t_{i-1}$

because t_i did not yet occur previously in the text immediately following the context $t_{i-k} \cdots t_{i-1}$, a new prediction is attempted with only $k-1$ symbols. This process is repeated until a context that already appeared previously in the text has been found, or no more symbols remain in the context, that is, $k = 0$ in case the character already appeared in the first $i-1$ symbols of T, or $k = -1$ in case the character is encountered in T for the first time.

An escape code, denoted by `$`, is used to inform the decoder to switch to a smaller context. The escape code is treated as a special character, and is assigned a probability within the distribution of the context it is sent. Different variants of PPM use different heuristics for its probability. In this paper we follow PPMC that sets the frequency of the escape code to the number of different characters seen in the given context.

As example, consider the text `lossless` and $k_{max} = 2$.

	0	1	2	3	4	5	6	7
T	l	o	s	s	l	e	s	s

At position 7 of T, when encoding the character `s`, we first consider its context of size $k_{max} = 2$, that is, the context `es` starting at position 5. This context has not occurred before position 7, therefore the encoder outputs an escape code `$` sign, informing the decoder that no content of length 2 has been found. Note that the probability for `$` is 1, as also the decoder realizes that the context `es` occurs for position 7 for the first time, and therefore, no bits are needed for the encoding of `$` in this case. The context is then shortened to the context of size 1, the context `s` at position 6. The context `s` appears before for the characters `s` and `l` at positions 3 and 4 with a single occurrence for each. As noted above, the PPMC variant sets the frequency for `$` to the number of different characters seen in the given context, which is 2 in our case. The distribution is therefore

$$p(\texttt{s}) = \frac{1}{4}, \quad p(\texttt{l}) = \frac{1}{4}, \quad p(\$) = \frac{1}{2}.$$

The character `s` at position 7 is therefore encoded in the context of `s` with probability $\frac{1}{4}$, using 2 bits. The *Prediction Frequency Table (PFT)* is then updated to include the contexts for `s`:

- for `es` - `s` and `$` with frequency 1;
- for `s` - `s` with frequency 2, `l` with frequency 1, and `$` with frequency 2;

Significant additional savings of the PPM encoding are achieved by using the *Exclusion Principle* (EP). The idea is that in case $k < k_{max}$, one can exclude all characters that appear in longer contexts than k, from appearing in the context of k. The reason for the exclusion of a character σ is that if σ does appear in a higher context than that indexed k, and σ is the following character to be encoded, it would have already been used at that higher context, and the context length would not have been shortened. As the encoder switched down to a shorter context, σ is not the following character to be encoded, and can therefore be removed from the current context. The EP enhancement increases the probabilities of the characters in the context of size k and thus the encoding may become more efficient.

Assume our running example is extended by `ly` to the text `losslessly`, and the encoder is about to encode the character `y` at position 9. The *PFT* is presented in

	0	1	2	3	4	5	6	7	8	9
T	l	o	s	s	l	e	s	s	l	y

Figure 4. The context of length $k_{max} = 2$ is now sl, which occurs also at position 3 as the context for e. As y does not appear in the context of sl, the character $ is encoded with probability $\frac{1}{2}$ (e and $ are both with frequency 1). Shortening the context to only l, the decoder already rules out the character e for being the following character to be decoded. The characters that appear with context l are o and e and occur once in this context. Thus, PPMC assigns the frequency 2 to $. After applying the EP, e was already ruled out by its appearance in a higher content, thus the original distribution

$$p(\mathtt{o}) = \frac{1}{4}, \quad p(\mathtt{e}) = \frac{1}{4}, \quad p(\$) = \frac{1}{2}$$

is changed to

$$p(\mathtt{o}) = \frac{1}{3}, \quad p(\mathtt{e}) = 0, \quad p(\$) = \frac{2}{3}.$$

The character $ is encoded and the context is shortened by 1. The empty context, $k = 0$, contains all characters that have already occurred in T, that is, l, o, s and e. By EP, characters e and o are ruled out, leaving only l, s and $ with frequencies 3, 4 and 4, respectively. $ is then encoded with probability $\frac{4}{11}$, and the context index changes to $k = -1$ where y is given the probability $\frac{1}{2}$ as all characters other than $ are excluded. Note that the $ is needed in context $k = -1$ for encoding EOF. The character y is encoded by $$$y with probabilities $\frac{1}{2}$, $\frac{2}{3}$, $\frac{4}{11}$ and $\frac{1}{2}$, instead of with probabilities $\frac{1}{2}$, $\frac{1}{2}$, $\frac{4}{13}$ and $\frac{1}{6}$ in case EP is not used. The entropy is reduced by 2.24 bits using EP.

The encoding algorithm with PPMC is given for self-containment. The PPMC decoding is symmetrical. Algorithm 1 presents the way each individual character t_i of T gets processed assuming a maximal size context of k_{max}. At a prefix $T[1, k_{max} - 1]$ of T when t_i gets processed, $i < k_{max}$, the parameter k_{max} of PPM_ENCODE is initialized by $i - 1$, the number of the already processed characters. Otherwise, k_{max} is the *order* of PPMC. The local variable *str* stores the current context, starting with the maximal sized context and progressively shortening it when t_i has never been seen in that context. If t_i has already been seen in the context of *str*, it is encoded in that probability distribution and the *PFT* is updated by incrementing the frequency of t_i in the *str* context. Otherwise, the $ sign is encoded in the context of *str*, the prediction table is updated and the length of *str* is shortened by eliminating its first character. The EP is implemented by eliminating all other characters that appear in context *str* from shorter contexts for t_i. For more details we refer the reader to [10].

4 PPM + Forward

As mentioned above, the Forward looking algorithm "looks into the future", and uses the information of what is still to come. At first sight it seems worthwhile to use as much knowledge as possible about the text, that is, the frequencies for each substring

Algorithm 1: *PPM Algorithm for encoding t_i of T.*

```
PPM_ENCODE(t_i, k_max)
  k ← k_max
  found ← false
  while k ≥ 0 and not found do
      str ← t_{i-k} ··· t_{i-1}
      // Exclusion Principle
      if ∃j and σ, k < j < k_max, so that freq(σ|t_{i-j} ··· t_{i-1}) > 0 then
          freq( σ|str) ← 0
       // t_i has been seen in context str
      if freq(t_i|str) > 0 then
          encode t_i based on context str
          found ← true
      else
          encode $ based on context str
          k ← k − 1
      // update PFT for future use
      if freq($|str)=0 then
          // context seen for the first time
          freq($|str)=freq(t_i|str)← 1
      else
          freq(t_i|str)++
          freq($|str)++
  if not found then
      encode t_i in the context of k = −1
```

of length $\leq k_{max}$. However, encoding this information implies an expensive storage overhead on the compressed form. The idea is thus to use only the information about the exact frequencies of the involved individual symbols, similarly to Forward, that may imply only a negligible overhead for huge files and fixed size alphabets, typically up to size 256.

The PPM algorithm uses the already processed portion of the file in order to predict the encoding of the current symbol, whereas Forward uses its knowledge of the future on what is still to come. The proposed algorithm integrates both strategies by using the past in order to predict the encoding, but utilizes its limited knowledge of the future in order to enhance the prediction. As the alphabet and the exact frequencies of the characters in the underlying text is known in advance, there is no need in the context referring to $k = -1$. Consequently, the escape code \$ is not required in the context of $k = 0$. Moreover, once the frequency of a certain character t_i is reduced to 0, t_i can be eliminated completely from PFT. However, this should be done with caution, as t_i can still occur in the context of characters that are less than k_{max} characters away from the current position. Formally, after processing t_i of T, $1 \leq i \leq n$, freq(t_i) is reduced by 1. In case freq(t_i)=0 we wish to remove t_i from all entries in the PFT that contain t_i, either in their context or as the predicted character. This elimination is performed gradually, and only at position $i + k_{max}$ all desired entries would effectively be removed. More precisely, if freq(t_i) becomes 0 after processing position i, t_i is removed from being the next predicted character (column σ in Figure 4). At position $i+1$, t_i can be removed from all entries that have t_i as the

last character of their context. Generally, the entry referring to context *str* can be deleted from PFT after processing position $i + j$ in case t_i occurs at the j-th position from its right.

Using our running example, Table 1 presents the contexts up to size k_{max} after the prefix `lossle` of T has been processed. The upper part of the table is the prediction made be PPM, while the lower part is the prediction table for PPM+Forward. Each main column corresponds to a different context size starting from $k = -1$, which is not applicable for PPM+Forward, and ending with $k = k_{max} = 2$. Each column is internally divided into sub-columns presenting the context, headed by con, a symbol $\sigma \in \Sigma$, and the frequency, freq, of σ within that context. As can be seen, the prediction for PPM+Forward is more accurate resulting in higher probabilities thus lower information content.

Size	$k=-1$			$k=0$			$k=1$			$k=k_{max}=2$		
	con	σ	freq	con	σ	freq	con	σ	freq	con	σ	freq
PPM	–	$\Sigma \cup \$$	1		l	2	l	o	1	lo	s	1
					e	1		e	1		$	1
				ϵ	s	2		$	2	os	s	1
					e	1	o	s	1		$	1
					$	4		$	1	ss	l	1
							s	s	1		$	1
								l	1	sl	e	1
								$	2		$	1
							l	e	1			
								$	1			
PPM+Forward	–	–	–		l	1		s	1	ss	l	1
				ϵ	s	2	s	l	1			
					y	1		$	1			

Figure 2. *Prediction frequencies for the prefix* `lossle` *of* $T =$ `losslessly` *for* PPM *and for* PPM+Forward.

5 Experimental Results

In order to evaluate the compression savings of the suggested PPM+Forward method relative to the original PPM compression algorithm, we have considered several datasets of different sizes and nature, and using different alphabets, independently and combined.

– *sources* is a C/Java source code file formed by the concatenation of .c, .h, .C and .java files of the linux-2.6.11.6 and gcc-4.0.0 distributions, downloaded from the Pizza & Chili Corpus[1] ;

[1] http://pizzachili.dcc.uchile.cl

– *ftxt* is the French version of the European Union's JOC corpus, a collection of pairs of questions and answers on various topics used in the ARCADE evaluation project [13];
– *etxt* is the translation of *ftxt* into English;
– *fe/ef* are the concatenations of *ftxt* and *etxt*, *fe* for *ftxt* before *etxt*, and *ef* in the other order.
– *nt* is a dataset constructed from the King James version of the English Bible in order to obtain a file composed of two parts having disjoint sets of alphabets — digits, followed by characters. The first part of the file takes the 10,000 first characters of the file and writes them as decimal digits; the second part consists of the remaining characters of Bible in their original form.

Table 1 summarizes the information regarding the used datasets.

File	full size (bytes)	$\|\Sigma\|$
sources	210,866,607	230
ftxt	7,648,930	132
etxt	6,611,031	125
fe/ef	14,259,963	134
nt	3,726,683	62

TABLE 1: *Information about the used datasets.*

File	Size of Encoded file (bytes)	
	PPM	PPM-FORWARD
sources	52,287,089	52,286,877
fe	4,955,525	4,954,399
ef	4,954,985	4,954,634
nt	968,147	968,023

TABLE 2: *Compression performance.*

Table 2 presents our experimental results with all figures given in bytes. The second column presents the size of the compressed file using the original PPM algorithm. The compression performance of our proposed method PPM+FORWARD is shown in the third column, where the size of the header is included in the reported numbers. For *sources* we used $k_{max} = 5$, and $k_{max} = 3$ for the other files. On the shown examples PPM+FORWARD slightly improves on PPM. For other examples, while the compressed file itself was smaller for PPM+FORWARD than for PPM alone, the addition of the header increased the size over that of pure PPM, so for these cases, it would not be worthwhile to apply the newly suggested combination of PPM with FORWARD.

An interesting fact to note is that the improvement for the file *fe* is more than 3 times as large as for *ef*. A possible explanation could be the following: French uses several accented and other characters, like é, à, û, ç, œ, etc. that do not appear in English texts, but this restriction is not symmetric. All English characters do appear in French texts, though not in the same contexts and with different probabilities, e.g., k is frequent in English but extremely rare in French. Therefore, once the French part of the *fe* file is passed, several characters can be eliminated from the PFT, already in about the middle of the file. However, for the *ef* file, the symmetric phenomenon does not occur, all the characters remain in the PFT and the savings are diminished.

6 Conclusion

Aiming at improving PPM, which is known as one of the best lossless compression schemes was pretentious to begin with. However, with a limited knowledge of the text we were able to present an enhancement, even though a quite moderate one, for certain types of files.

References

1. J. Cleary and I. Witten: *Data compression using adaptive coding and partial string matching.* IEEE Transactions on Communications, 32(4) 1984, pp. 396–402.
2. P. Elias: *Universal codeword sets and representations of the integers.* IEEE Trans. Information Theory, 21(2) 1975, pp. 194–203.
3. A. Fruchtman, Y. Gross, S. T. Klein, and D. Shapira: *Weighted adaptive coding.* CoRR, abs/2005.08232 2020.
4. A. Fruchtman, S. T. Klein, and D. Shapira: *Bidirectional adaptive compression*, in Proceedings of the Prague Stringology Conference 2019, 2019, pp. 92–101.
5. D. A. Huffman: *A method for the construction of minimum-redundancy codes.* Proceedings of the IRE, 40(9) 1952, pp. 1098–1101.
6. S. T. Klein, S. Saadia, and D. Shapira: *Forward looking Huffman coding*, in The 14th Computer Science Symposium in Russia, CSR, Novosibirsk, Russia, July 1-5, 2019.
7. S. T. Klein and D. Shapira: *A new compression method for compressed matching*, in Data Compression Conference, DCC 2000, Snowbird, Utah, USA, 2000, pp. 400–409.
8. S. T. Klein and D. Shapira: *On improving Tunstall codes.* Inf. Process. Manage., 47(5) 2011, pp. 777–785.
9. S. T. Klein and D. Shapira: *Random access to Fibonacci encoded files.* Discrete Applied Mathematics, 212 2016, pp. 115–128.
10. A. Moffat and A. Turpin: *Compression and Coding Algorithms*, Springer US, 2002.
11. J. A. Storer and T. G. Szymanski: *Data compression via textural substitution.* J. ACM, 29(4) 1982, pp. 928–951.
12. B. Tunstall: *Synthesis of Noiseless Compression Codes*, Georgia Institute of Technology, 1967.
13. J. Véronis and P. Langlais: *Evaluation of parallel text alignment systems: The* arcade *project*, in Parallel Text Processing, J. Véronis, ed., Kluwer Academic Publishers, Dordrecht, Chapter 19, 2000, pp. 369–388.
14. J. S. Vitter: *Algorithm 673: Dynamic Huffman coding.* ACM Trans. Math. Softw., 15(2) 1989, pp. 158–167.

15. T. Welch: *A technique for high-performance data compression.* IEEE Computer, 17(6) 1984, pp. 8–19.

16. I. H. Witten, R. M. Neal, and J. G. Cleary: *Arithmetic coding for data compression.* Commun. ACM, 30(6) 1987, pp. 520–540.

17. J. Ziv and A. Lempel: *A universal algorithm for sequential data compression.* IEEE Transactions on Information Theory, 23(3) 1977, pp. 337–343.

18. J. Ziv and A. Lempel: *Compression of individual sequences via variable-rate coding.* IEEE Transactions on Information Theory, 24(5) 1978, pp. 530–536.