SN Computer Science (2022) 3:239
https://doi.org/10.1007/s42979-022-01121-0

ORIGINAL RESEARCH

Combining Forward Compression with PPM

Rachel Mustakis Avrunin[1] · Shmuel T. Klein[2] · Dana Shapira[1]

Received: 10 November 2021 / Accepted: 30 March 2022 / Published online: 23 April 2022

Abstract
A new Forward Looking variant of dynamic Huffman or arithmetic encoding has been recently proposed, that provably always performs better than the corresponding general static encoding schemes, as far as the net compressed file, without the necessary header, is concerned. The current paper suggests to integrate the Forward Looking paradigm with the well-known adaptive PPM—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 empirically shown to enhance the prediction capability, and therefore to improve the compression efficiency.

Keywords Lossless compression · Arithmetic coding · PPM

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 Prediction by Partial Matching (PPM) [1] algorithm with a recently introduced compression paradigm named Forward Looking, which is based on statistical coding such as Huffman [2] 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 under certain constraints, while still being simple. Given is a text T of size 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 the occurrence 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 $H = -\sum_{i=1}^{m} p_i \log p_i$, called the *entropy*. This average can be reached by applying *arithmetic coding* [3]. Note that the term *alphabet* should be understood in a broader sense, as its elements are not restricted to merely standard characters and may consist of strings or words, as long as there is a well-defined way to break T into a sequence of elements of Σ.

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 [2], Elias [4], and Fibonacci coding [5]. 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 [6, 7].

This article is part of the topical collection "String Processing and Combinatorial Algorithms" guest edited by Simone Faro.

Rachel Mustakis Avrunin and Shmuel T. Klein are contributing authors.

✉ Dana Shapira
shapird@g.ariel.ac.il

Rachel Mustakis Avrunin
r.avrunin@gmail.com

Shmuel T. Klein
tomi@cs.biu.ac.il

[1] Department of Computer Science, Ariel University, Ramat HaGolan St. 65, 40700 Ariel, Israel

[2] Department of Computer Science, Bar Ilan University, 52900 Ramat-Gan, Israel

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 was introduced by Ziv and Lempel, in particular, LZ77 [8] and LZ78 [9]. Storer and Szymanski [10] proposed LZSS a practical variant of the LZ77 compressor, which is based on the principle of finding redundant strings or patterns and replacing them by pointers to a common copy. LZW [11] is an implementation of LZ78, in which 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.

PPM is yet another adaptive coding method, proposed by Cleary et al. [1] as a combination of statistical coding with a Markov model, and it 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 [12]. Despite its space efficiency, the technique suffers from slow processing time, and therefore, 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 [13]. This is achieved by redefining the reference pointers of the encoded elements to those forming the model of the encoding; the new definition reverses the direction of the pointers 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 Ref. [14], taking both past and future into account, and its net encoding is provably at least as good as the variant based solely on the future.

The current paper suggests to integrate the Forward Looking paradigm with PPM. 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, enhances the prediction capability on our tests, and therefore improves the compression efficiency. Our paper is organized as follows. "Forward Looking Dynamic Huffman Coding" and "PPM Compression" sections recall the details of Forward looking and PPM, respectively. "PPM + Forward Compression" section combines the two algorithms, and presents a new implementation of the PPM algorithm that is based on Forward.

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 assumes a uniform distribution over the given alphabet. The model for static codes may be more advanced and assigns a fixed, not necessarily uniform, 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 (and not including) that position. In case the exact number of occurrences of each element in the entire file is known, e.g., when this information can be obtained by a preprocessing scan of the file, a different, Forward looking adaptive approach can be applied [13]. 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 Ref. [15] for performing streaming pattern matching in LZSS 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 increment its frequency, implying a decrease in its information content. However, consequently, the information content of certain other elements may increase, and thus also the entropy of the entire text. The forward looking paradigm provides a more "social" approach that takes all the elements into account, rather than only the one that is currently being processed: the frequency 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. To distinguish between the backward and forward approaches, we use the notations b-freq to refer to the (backward) frequencies of individual characters up to (and not including) the current point, and f-freq to refer to (forward) frequencies—the frequencies of each character from (and including) the current point onward, up to the end of the file.

Table 1 The information content of Forward Looking as compared to static and adaptive on $T = \texttt{lossless}$

i	t_i	static		adaptive		Forward	
		$p(t_i)$	$-\log(p(t_i))$	$p(t_i)$	$-\log(p(t_i))$	$p(t_i)$	$-\log(p(t_i))$
0	l	1/4	2.000	1/4	2.000	1/4	2.000
1	o	1/8	3.000	1/5	2.322	1/7	2.807
2	s	1/2	1.000	1/6	2.585	2/3	0.585
3	s	1/2	1.000	2/7	1.807	3/5	0.737
4	l	1/4	2.000	1/4	2.000	1/4	2.000
5	e	1/8	3.000	1/9	3.170	1/3	1.585
6	s	1/2	1.000	3/10	1.737	1.0	0.000
7	s	1/2	1.000	4/11	1.459	1.0	0.000
Total			14.000		17.080		9.714

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 to be more efficient than the dynamic Huffman algorithm of Vitter [16], since it relies on known statistics instead of learning them "on the fly". This 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 [17]). The Forward looking approach has been extended to weighted forward coding in Ref. [17], and to a backward weighted heuristic in Ref. [18].

Arithmetic coding represents the text T by a real number a in the range $0 \le a < 1$. An interval $[\ell, h)$ is initialized by [0, 1), and is partitioned into sub-intervals according to the probability distribution of the characters. The procedure continues with the sub-interval selected according to the currently processed symbol of T. Therefore, 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.

The general Forward algorithm initializes its model identically to the standard static version. That is, the Huffman implementation of Forward starts with the same Huffman tree as the static variant, 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 used, 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, the general step of Forward consists of the two following commands:

1. encode the next element based on the current model;
2. update the model by decrementing the (forward) frequency f-freq of the current element and adjusting the distribution accordingly.

Example

Forward

As example, consider the text $T = \texttt{lossless}$. The alphabet Σ is $\{\texttt{e}, \texttt{l}, \texttt{o}, \texttt{s}\}$ with frequencies $\{1, 2, 1, 4\}$, respectively. We denote the distribution of the forward, static and adaptive encodings after having processed the first i characters of the text by P_i^{f}, P_i^{s} and P_i^{a}, respectively, so that the initial distribution for forward is given by

$$P_0^{\mathsf{f}} = \left\{ p_0^{\mathsf{f}}(\texttt{e}) = \tfrac{1}{8}, \quad p_0^{\mathsf{f}}(\texttt{l}) = \tfrac{1}{4}, \quad p_0^{\mathsf{f}}(\texttt{o}) = \tfrac{1}{8}, \quad p_0^{\mathsf{f}}(\texttt{s}) = \tfrac{1}{2} \right\},$$

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

$$P_1^{\mathsf{f}} = \left\{ p_1^{\mathsf{f}}(\texttt{e}) = \tfrac{1}{7}, \quad p_1^{\mathsf{f}}(\texttt{l}) = \tfrac{1}{7}, \quad p_1^{\mathsf{f}}(\texttt{o}) = \tfrac{1}{7}, \quad p_1^{\mathsf{f}}(\texttt{s}) = \tfrac{4}{7} \right\}.$$

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

$$P_2^{\mathsf{f}} = \left\{ p_2^{\mathsf{f}}(\texttt{e}) = \tfrac{1}{6}, \quad p_2^{\mathsf{f}}(\texttt{l}) = \tfrac{1}{6}, \quad p_2^{\mathsf{f}}(\texttt{s}) = \tfrac{2}{3} \right\}.$$

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

$$P_3^{\mathsf{f}} = \left\{ p_3^{\mathsf{f}}(\texttt{e}) = \tfrac{1}{5}, \quad p_3^{\mathsf{f}}(\texttt{l}) = \tfrac{1}{5}, \quad p_3^{\mathsf{f}}(\texttt{s}) = \tfrac{3}{5} \right\},$$

and so on. Note that when only a single symbol remains in the text to be processed, no additional encoding is needed, because the decoder also realizes this case. Therefore, the last two symbols s are not encoded.

Table 1 is a comparative chart to highlight the differences between static, adaptive, and forward arithmetic coding, showing for each character t_i of T the currently used probability $p(t_i)$ and the corresponding information content $-\log(p(t_i))$. The last line of the table shows the total number of bits used by each method. This is in fact the information content of choosing a real number in the final interval.

static

Continuing with our running example, the initial interval [0, 1) for the static variant is partitioned, like for the forward case, according to

$$P^{s}_{0} = \left\{p^{s}_{0}(\texttt{e}) = \tfrac{1}{8},\quad p^{s}_{0}(\texttt{l}) = \tfrac{1}{4},\quad p^{s}_{0}(\texttt{o}) = \tfrac{1}{8},\quad p^{s}_{0}(\texttt{s}) = \tfrac{1}{2}\right\},$$

for example, choosing lexicographic order. This yields the partition of [0, 1) into sub-intervals $[0, \frac{1}{8})$, $[\frac{1}{8}, \frac{3}{8})$, $[\frac{3}{8}, \frac{1}{2})$ and $[\frac{1}{2}, 1)$ for e, l, o and s, respectively. The first step is then to select the sub-interval $[\frac{1}{8}, \frac{3}{8})$ referring to l, which is the first character of T. The size of this interval is $\frac{1}{4}$, corresponding to an information content of 2 bits. The current interval is narrowed by a factor of 1/8 which is the probability of the next character o and adds 3 bits to the size of the encoded file. Generally, the number of bits added to the encoded file by each processed character is exactly its information content.

adaptive

Unlike the static coding, in which the partition of [0, 1) into sub-intervals for arithmetic coding, or the tree for Huffman coding, is fixed throughout the process, the dynamic variants update this partition or the tree 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 for arithmetic, or the weights of all the leaves for Huffman, are adjusted accordingly.

For the adaptive variant, the initial partition of the interval [0, 1) for our running example is divided into four sub-intervals of equal size $\frac{1}{4}$, suiting the uniform distribution, and the first character l is encoded by 2 bits. The (backward) frequency of l, b-freq(l), is then incremented by 1, and the distribution is updated to

$$P^{a}_{1} = \left\{p^{a}_{1}(\texttt{e}) = \tfrac{1}{5},\quad p^{a}_{1}(\texttt{l}) = \tfrac{2}{5},\quad p^{a}_{1}(\texttt{o}) = \tfrac{1}{5},\quad p^{a}_{1}(\texttt{s}) = \tfrac{1}{5}\right\},$$

and the next character o is encoded with probability $\frac{1}{5}$. The frequency of o, b-freq(o), is then incremented, and the distribution is updated to

$$P^{a}_{2} = \left\{p^{a}_{2}(\texttt{e}) = \tfrac{1}{6},\quad p^{a}_{2}(\texttt{l}) = \tfrac{1}{3},\quad p^{a}_{2}(\texttt{o}) = \tfrac{1}{3},\quad p^{a}_{2}(\texttt{s}) = \tfrac{1}{6}\right\}.$$

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 second set of columns of Table 1 headed by adaptive.

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 given Y, $H(X|Y)$, is at most the entropy of X, $H(X)$, that is

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

The entropy $H = \sum_{i=1}^{m} -p_i \log p_i$ is the expected number of bits to encode the following character; therefore, there is a gain when we take a context into account. The parameter $k_{\max}$ is defined as the maximum allowed context size, that is, the number of examined previous characters $t_{i-k_{\max}} \cdots t_{i-1}$, that are used to predict and encode the current character t_i. $k_{\max}$ is also named the *order* of PPM, and is typically less or equal to 8; otherwise, the space for storing all the details of the model becomes too large to be handled in RAM.

The process for encoding each symbol t_i, $0 \leq i < n$, is initialized by assigning $k = k_{\max}$. If no prediction for t_i can be made based on its k preceding 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 $t_{i-(k-1)} \cdots t_{i-1}$. 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 codeword, denoted by \$, is used to inform the decoder to switch to a smaller context. The escape codeword 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 codeword to the number of different characters seen in the given context.

We continue with our running example $T = t_0 \cdots t_7$

i	0	1	2	3	4	5	6	7
t_i	`l`	`o`	`s`	`s`	`l`	`e`	`s`	`s`

with $k_{\max} = 2$, where the small numbers above the text are the position indices. 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 symbol `$`, informing the decoder that no context of length 2 has been found. Note that the probability for `$` is 1, as also the decoder realizes that the context `es` occurs at position 7 for the first time, and therefore, no bits are needed for the encoding of `$` in this case. The context is then shortened to size 1, which is the single character `s` at position 6. The context `s` appears before for the characters `s` and `l` at positions 3 and 4 with a single occurrence for each. As noted above, the PPMC variant sets the frequency of `$` to the number of different characters seen in the given context, which is 2 in our case. The distribution in the context $t_6 =$ `s`, after having processed the 7 first characters `lossles`, is therefore

$$p_7^{\mathsf{PPM}}(\mathtt{s}) = \tfrac{1}{4}, \quad p_7^{\mathsf{PPM}}(\mathtt{l}) = \tfrac{1}{4}, \quad p_7^{\mathsf{PPM}}(\$) = \tfrac{1}{2}.$$

The character `s` at position 7 is thus encoded in the context `s` with probability $\frac{1}{4}$, using 2 bits. The *Prediction Frequency Table* (PFT) is 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 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.

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

```
PPM_Encode($i, k_{\max}$)
$k \leftarrow k_{\max}$
found $\leftarrow$ false
$\mathcal{L} \leftarrow \emptyset$
while $k \geq 0$ and not found do
    $str \leftarrow t_{i-k} \cdots t_{i-1}$
    // Exclusion Principle
while $\exists(j, \sigma)$ so that $k < j \leq k_{\max}$ and b-freq($\sigma \mid t_{i-j} \cdots t_{i-1}$) $> 0$ then
        $\mathcal{L} \leftarrow \mathcal{L} \cup \{\sigma\}$
    end if
    // $t_i$ has been seen in context $str$
    if b-freq($t_i \mid str$) $> 0$ then
        encode $t_i$ based on context $str$, excluding elements of $\mathcal{L}$
        found $\leftarrow$ true
    else
        encode $ based on context $str$, excluding elements of $\mathcal{L}$
        $k \leftarrow k - 1$
    end if
    // update PFT for future use
    if b-freq($ $\mid str$) $= 0$ then
        // context seen for the first time
        b-freq($ $\mid str$) $\leftarrow 1$
        b-freq($t_i \mid str$) $\leftarrow 1$
    else
        b-freq($t_i \mid str$)++
        b-freq($ $\mid str$)++
    end if
end while
if not found then
    encode $t_i$ in the context of $k = -1$
end if
```

Assume our running example is extended by `ly` to the text `losslessly` indexed 0 to 9, and the encoder is about to encode the character `y` at position 9. The context of length $k_{\max} = 2$ is `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 `$` are both with frequency 1). Shortening the context to only the character `l`, the decoder already rules out the character `e` for being the following character to be decoded. The characters that appear with context `l` are `o` and `e` and occur once in this context. Thus, PPMC assigns the (backward) frequency 2 to `$`. After applying the EP, `e` is already ruled out by its appearance in a higher context; thus, the original distribution

$$p_9^{\mathsf{PPM}}(\mathtt{o}) = \tfrac{1}{4}, \quad p_9^{\mathsf{PPM}}(\mathtt{e}) = \tfrac{1}{4}, \quad p_9^{\mathsf{PPM}}(\$) = \tfrac{1}{2}$$

is changed to

$$p_9^{\mathsf{PPM+EP}}(\mathtt{o}) = \tfrac{1}{3}, \quad p_9^{\mathsf{PPM+EP}}(\mathtt{e}) = 0, \quad p_9^{\mathsf{PPM+EP}}(\$) = \tfrac{2}{3}.$$

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 $\frac{1}{2}$, because 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 $k_{\max}$. It uses a local list $\mathcal{L}$ to store the characters on which the EP applies. At a prefix $T[1, k_{\max} - 1]$ of T when t_i gets processed for $i < k_{\max}$, 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 according to that probability distribution and the PFT is updated by incrementing the frequency, b-freq, of t_i in the *str* context, denoted by b-freq$(t_i \mid str)$. Otherwise, the `$` sign is encoded in the context of *str*, the prediction frequency table is updated and the length of *str* is shortened by eliminating its leftmost 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 [12].

PPM + Forward Compression

As mentioned above, the Forward looking paradigm “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 of all characters for each possible context of length $\le k_{\max}$. However, encoding this information implies a storage overhead on the compressed form that might be prohibitive in terms of the amount of RAM that can still be handled efficiently. The extra information needed for merely pairs of characters is already a factor of $|\Sigma|$ larger than that needed for single characters, which is too expensive to justify the forward looking approach. The idea is thus to use only the information about the global frequencies of the individual symbols (context of size 0), similarly to Forward, that may imply only a negligible overhead for large files and fixed size alphabets, typically up to size 256.

The proposed algorithm integrates both strategies: it uses the past, like PPM, to predict the current symbol, and utilizes its limited knowledge of the future, as in Forward, to enhance the prediction. As the alphabet and the exact frequencies of the characters in the underlying text are known in advance, there is no need in the context referring to $k = -1$.

Consequently, the escape symbol `$` is not required for $k = 0$. The frequencies in a context of size 0 are updated to reflect the number of occurrences f-freq of each symbol in the remaining portion of the file, while the frequencies b-freq of characters in higher contexts correspond to the number of occurrences seen in the already processed portion of the file up to the current point. After processing t_i of T, f-freq(t_i) in a context of size 0 is reduced by 1 according to the forward looking paradigm. In case f-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 must be performed gradually, and only at position $i + k_{\max}$, all relevant entries would effectively be removed.

More precisely, in the special case when f-freq(t_i) in context 0 becomes 0 after processing position i, t_i is removed from being the next predicted character in the columns headed σ in the PFT. After processing position $i + 1$, the entries of the PFT that have t_i as the rightmost character of their context can be removed. Generally, an entry referring to context *str* can be deleted from PFT after processing position $i + j$, in case t_i occurs at the jth position from the right of *str*, $1 \le j \le \min(|str|, k_{\max})$.

Using our running example, Table 2 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 by PPM, while the lower part is the prediction frequency table for PPM+F. Each main column corresponds to a different context size starting from $k = k_{\max} = 2$ and ending with $k = -1$, which is not applicable for PPM+F. Each column is internally divided into sub-columns presenting the context, headed by con, a symbol $\sigma \in \Sigma$, and the frequency of σ within that context (headed by b-freq, except for context of size 0 of PPM+F that corresponds to f-freq). As can be seen, the prediction for PPM+F is more accurate on this example, resulting in lower average information content.

Discussion

For an input file T to be encoded, let T_{PPM} and $T_{\text{PPM+F}}$ denote the actual symbols that are produced by PPM and PPM+F, respectively, i.e., including the escape codewords denoted by

Table 2 PFT: Prediction frequencies after having processed the prefix `lossle` of $T =$ `lossle ssly` for PPM and for PPM+F

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

For each context con of size k, $-1 \le k \le 2$, the possible characters σ and their (backward) frequencies are listed

`$`, which have been adjoined to T. In fact, T_{PPM} and $T_{\mathsf{PPM+F}}$ differ only for the following two cases:

1. there is a missing `$` in $T_{\mathsf{PPM+F}}$ whenever T_{PPM} uses the context of size -1;
2. the last appearance of a symbol is not coded in $T_{\mathsf{PPM+F}}$, and in particular, the last symbol of T, or a run of identical last symbols, is not coded.

For $0 < k \le k_{\max}$, the distribution of the symbols in the context of size k for PPM+F is the same as for PPM, except for when symbols start to disappear. We consider the distribution at position i of $T_{\mathsf{PPM+F}}$, and partition the characters that occur in the context of size k into two subsets:

- $\mathcal{X}_i$ is the set of characters that do not occur in the remaining portion of $T_{\mathsf{PPM+F}}$;
- $\mathcal{Y}_i$ is the set of characters that will still occur in the remaining portion of $T_{\mathsf{PPM+F}}$;

At position i of $T_{\mathsf{PPM+F}} = s_1 s_2 \cdots$, the current character s_i is encoded using the probability for s_i according to the distribution of the maximum possible current context of size k, following the escape code for longer contexts, if any. The probability of an occurrence of s_i according to PPM is

$$p_1 = \frac{\text{b-freq }(s_i)}{\sum_{x\in\mathcal{X}_i} \text{b-freq }(x) + \sum_{y\in\mathcal{Y}_i} \text{b-freq }(y) + |\mathcal{X}_i| + |\mathcal{Y}_i|},$$

where the denominator sums the frequencies of all the characters of Σ, and $|\mathcal{X}_i|$ and $|\mathcal{Y}_i|$ are added as the frequency of the escape symbol, while the probability of an occurrence of s_i according to PPM+F for context $k > 0$ is

$$p_2 = \frac{\text{b-freq }(s_i)}{\sum_{y\in\mathcal{Y}} \text{b-freq }(y) + |\mathcal{Y}_i|},$$

as above, but including only the characters in $\mathcal{Y}_i$ that will still appear. Obviously, $p_1 \le p_2$, implying that for contexts of size $k > 0$

$$\begin{aligned}&\text{the information content of } s_i \text{ by PPM}\\ &\text{cannot be smaller than that for PPM + F.}\end{aligned} \tag{1}$$

The additional symbols that only occur in T_{PPM} and not in $T_{\mathsf{PPM+F}}$ may only increase the size of the compressed file by PPM.

Imagine an intermediate algorithm, inter-Algo, that for context $k = 0$, uses the backward frequencies as done by PPM+F for contexts $k > 0$, instead of the forward frequencies. That is, inter-Algo operates just as PPM, except that it is also given the exact frequencies f-freq of the characters of the input file, and it removes irrelevant entries from PFT when possible, similarly to PPM+F.

Obviously, the claim in (1) for $k > 0$ can be extended to $k = 0$, resulting in

$$\begin{aligned}&\text{the net size of the inter-Algo encoded file}\\ &\text{is no more than that of PPM encoding,}\end{aligned} \tag{2}$$

where, here and below, the net size refers to the compressed file only, excluding the header with the frequencies. Our motivation for the PPM+F algorithm is the theorem proved in Ref. [19] that for a given input file and adaptive coding

$$\text{the net size of the Forward encoded file is less than that of the backward encoding}. \tag{3}$$

Therefore, we replace the backward frequencies of context $k = 0$ with the forward ones to enhance the compression efficiency. We do not know if claim 3 can be extended to showing that PPM+F is always better than PPM, even though our empirical results support this claim, as can be seen in "Conclusion" section.

In any case, even if the claim holds for the entire file, it is not necessarily true that *each* codeword of PPM+F, at every position, is of equal length or shorter than the corresponding codeword of the PPM code, as illustrated in the following subsection.

Examples

Consider as example the file $T = \mathtt{ab}^{1023}$. The traditional PPM uses $\log_2 3$ bits to encode the first a, as the probability for the context corresponding to $k = -1$ is 1/3 for the alphabet of size 2 and the adjoined escape symbol. On the other hand, PPM+F uses the context 0 with frequencies 1 and 1023 for a and b, respectively, encoding the a by 10 bits, for its probability at the first point of the file is 1/1024. This gap vanishes when the bs are encoded, no matter what parameter $k_{\max}$ is used. While PPM+F has all the information needed to encode the run of 1023 bs without any additional bits, the traditional PPM only then starts learning the distribution. As example assume that $k_{\max} = 1$. In this case, the encoding of two \$s follow the encoding of the a, with probabilities 1 and 1/2 for context sizes 1 and 0, respectively, because in context 0, a appears in addition to the \$ symbol. The first b then uses the probability 1/3 with $\log_2 3$ bits. The second b is encoded by an escape code for context 1 with probability 1, as it is the first time that b appears in the context of b. The b is then encoded with probability 1/4 as the distribution for context 0 is

$$p_1^{\mathsf{PPM}}(\mathtt{a}) = \frac{1}{4}, \quad p_1^{\mathsf{PPM}}(\mathtt{b}) = \frac{1}{4}, \quad p_1^{\mathsf{PPM}}(\$) = \frac{2}{4}.$$

From this point onward, only the context of size 1 is used for the encoding of the following bs with probabilities 1/2, 2/3, 3/4,..., 1021/1022 while increasingly more bs are encountered in the context of b. The final EOF then uses three \$ signs with probabilities 1/1023, 2/1026, and 1/3, in which the first two are used to escape from context of size 1 and from context of size 0.

Interestingly, if T is the text $\mathtt{ab}^{1022}\mathtt{a}$, the bs must be encoded by PPM+F. Both algorithms operate identically for encoding the bs in context 1, starting from the third appearance of b with probabilities 1/2, 2/3, 3/4, …, 1020/1021. While PPM starts as in the previous example, PPM+F encodes \$ and a in contexts of sizes 1 and 0 with probabilities 1 and 2/1024, the latter with occurrences a=2 and b=1022 (as \$ does not occur in the empty context). Next, \$, b, \$ and b are encoded in contexts of sizes 1, 0, 1, and 0 with probabilities 1, 1022/1023, 1, and 1021/1022, respectively. The last a is not encoded by PPM+F, but is encoded by PPM as \$ and a in contexts of sizes 1 and 0 and probabilities 1/1022 and 1/1025. The EOF is encoded again by three \$ signs and probabilities 1/1022, 2/1026, and 1/3, for contexts of sizes 1, 0, and −1. Obviously, the 10 bits for the encoding the first a by PPM+F and the encodings of the bs in context of size 1 are approximately balanced by the encoding of the two escape symbols for the EOF and the encodings of the bs in context of size 1 by PPM. Nevertheless, the remaining encodings only belong to PPM, implying a larger compressed file.

Next, we propose a hybrid approach that combines PPM and Forward Looking, but tries to overcome the expensive overhead of transmitting the entire set of character frequencies. We consider a sequence of subsets $\mathcal{S}$ of Σ of increasing size, starting with the empty set, and ending with the entire alphabet Σ. Thus, the traditional PPM corresponds to the alphabet of size 0, and PPM+F corresponds to the alphabet of size Σ. The hybrid approach requires the context corresponding to $k = -1$, used when a character is encountered for the first time, because only a partial alphabet is known. We then use the uniform distribution, as in the traditional PPM, over the alphabet of size $|\Sigma| - |\mathcal{S}|$.

It turns out that the compression efficiency is not correlated with the number of individual characters used in the PPM+F algorithm. That is, for a given subset $\mathcal{S}$ of Σ, the compression efficiency of PPM+F given the knowledge of the frequencies of the characters in $\mathcal{S}$ does not necessarily improve as $\mathcal{S}$ becomes larger. One could have expected that as the size of the subset $\mathcal{S}$ of Σ increases, the better the compression efficiency gets. However, the following example shows that this is not true.

Let T be a text over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$ with frequencies 1, 5, 16, and 2, respectively, and assume a prefix cd of T. We encode T using PPM+F with $k_{\max} = 1$, and we assume that we start with $\mathcal{S} = \emptyset$ and that the characters a, b, c and d are adjoined to $\mathcal{S}$ in this (lexicographic) order.

For $\mathcal{S} = \{\mathtt{a}\}$, only the frequency for a is known. When c, the first character of T, is processed, the distribution according to the current knowledge is $P_0^{\mathsf{PPM+F}} = \left\{ p_0^{\mathsf{PPM+F}}(\mathtt{a}) = \frac{1}{2}, \quad p_0^{\mathsf{PPM+F}}(\$) = \frac{1}{2} \right\}$, and the character c at position 0 is encoded by two \$s with probabilities 1 and $\frac{1}{2}$, respectively, since c does not occur in the context of length $k = 1$, nor in the context of length $k = 0$. These two \$s are followed by the encoding of c, with probability $\frac{1}{4}$ in the context of size −1. The information content is 0, 1, and 2 bits for \$, \$ and c, respectively, for a total of 3 bits. The distribution for the empty context is updated to $P_1^{\mathsf{PPM+F}} = \left\{ p_1^{\mathsf{PPM+F}}(\mathtt{a}) = \frac{1}{4}, \quad p_1^{\mathsf{PPM+F}}(\mathtt{c}) = \frac{1}{4}, \right.$ $\left. p_1^{\mathsf{PPM+F}}(\$) = \frac{1}{2} \right\}$, and the character d at position 1 is encoded again

Table 3 Example of the hybrid PPM + Forward Looking for $T = \texttt{cd}\cdots$, comparing alphabet subsets $\mathcal{S} = \{\texttt{a}\}$ and $\mathcal{S} = \{\texttt{a}, \texttt{b}\}$

	$\mathcal{S} = \{\texttt{a}\}$			$\mathcal{S} = \{\texttt{a}, \texttt{b}\}$		
	$(p_\texttt{a}, p_\texttt{b}, p_\texttt{c}, p_\texttt{d}, p_\$)$	p	$-\log(p)$	$(p_\texttt{a}, p_\texttt{b}, p_\texttt{c}, p_\texttt{d}, p_\$)$	p	$-\log(p)$
\$	(0,0,0,0,1)	1	0.000	(0,0,0,0,1)	1	0.000
\$	$(\frac{1}{2}, 0, 0, 0, \frac{1}{2})$	1/2	1.000	$(\frac{1}{8}, \frac{5}{8}, 0, 0, \frac{1}{4})$	1/4	2.000
c	$(-, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4})$	1/4	2.000	$(-, -, \frac{1}{3}, \frac{1}{3}, \frac{1}{3})$	1/3	1.585
\$	(0, 0, 0, 0, 1)	1	0.000	(0,0,0,0,1)	1	0.000
\$	$(\frac{1}{4}, 0, \frac{1}{4}, 0, \frac{1}{2})$	1/2	1.000	$(\frac{1}{10}, \frac{1}{2}, \frac{1}{10}, 0, \frac{3}{10})$	3/10	1.736
d	$(-, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4})$	1/4	2.000	$(-, -, \frac{1}{3}, \frac{1}{3}, \frac{1}{3})$	1/3	1.585
Total		6.000			6.906	

The leftmost column shows the coded characters, the other columns give the probability distributions, the probability of the encoded character, and the corresponding information content in bits

by two \$s followed by d, with the same probabilities and, consequently, the same information content of 3 bits.

For $\mathcal{S} = \{\texttt{a}, \texttt{b}\}$, the frequency for b is also known in addition to the frequency for a. When c is processed, the known distribution for the context of size 0 is $P_0^{\text{PPM+F}} = \left\{ p_0^{\text{PPM+F}}(\texttt{a}) = \frac{1}{8},\ \ p_0^{\text{PPM+F}}(\texttt{b}) = \frac{5}{8},\ \ p_0^{\text{PPM+F}}(\$) = \frac{1}{4} \right\}$, and the character c at position 0 is encoded by two \$s with probabilities 1 and $\frac{1}{4}$, as c does not occur in the contexts of length $k = 1$ and $k = 0$, followed by the encoding for c, with probability $\frac{1}{4}$. The information content is 0, 2, and 2 bits for both \$s and c, respectively, for a total of 4 bits, which is one bit more than for the case $\mathcal{S} = \{\texttt{a}\}$. The next character d at position 1 is encoded by two \$s followed by d, with probabilities, 1, $\frac{3}{10}$, and $\frac{1}{3}$, with information content 0, 1.736 and 1.585, as the distribution for the empty context is revised to

$$\left\{ p_1^{\text{PPM+F}}(\texttt{a}) = \frac{1}{10},\quad p_1^{\text{PPM+F}}(\texttt{b}) = \frac{1}{2},\quad p_1^{\text{PPM+F}}(\texttt{c}) = \frac{1}{10},\quad p_1^{\text{PPM+F}}(\$) = \frac{3}{10} \right\}.$$

The total information content, 3.321, is again larger than the information content for the case of $\mathcal{S} = \{\texttt{a}\}$. Table 3 summarizes this example, showing that for $\mathcal{S} = \{\texttt{a}, \texttt{b}\}$, we need 6.906 bits, while for $\mathcal{S} = \{\texttt{a}\}$, we need 6 bits. The example illustrates that more information does not necessarily reduce the size of the encoding.

Experimental Results

To evaluate the compression savings of the suggested PPM+F method relative to the original PPM compression algorithm, we have considered severaldatasets of different sizes and nature, and using different alphabets. The last six datasets have been 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 [20];
- *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;
- *XML* provides bibliographic information on major computer science publications, obtained from dblp.uni-trier.de;
- *dna* is a sequence of gene DNA sequences obtained from the Gutenberg Project;
- *english* is the concatenation of English text files selected from the Gutenberg Project;
- *pitches* is a sequence of pitch values (bytes in 0-127, plus a few extra special values) obtained from a myriad of MIDI files freely available on Internet;
- *proteins* is a sequence of protein sequences obtained from the Swissprot database; and

Table 4 Compression performance with $k_{\max} = 3$

File			Size of Encoded file (bytes)		
	Size	$\lvert\Sigma\rvert$	PPM	PPM-Forward	
				net-encoding	*total*
ftxt	7,648,930	132	**1,973,912**	1,973,801	1,974,049
etxt	6,611,031	125	1,628,998	1,628,596	**1,628,870**
ef	14,259,961	134	3,880,801	3,880,232	**3,880,500**
fe	14,259,961	134	3,881,723	3,881,429	**3,881,697**
XML	4,194,304	91	**666,194**	665,985	666,215
dna	52,428,800	16	12,581,286	12,581,235	**12,581,275**
english	52,428,800	176	15,858,858	15,856,738	**15,857,114**
pitches	4,194,304	125	2,118,754	2,117,892	**2,118,185**
proteins	52,428,800	25	26,643,210	26,643,105	**26,643,199**
sources	4,194,304	99	1,105,969	1,105,560	**1,105,839**

For every test file, the best performance is emphasized

– *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.

The first three columns of Table 4 summarize the information regarding the used datasets. The last three columns present the compression results given in bytes for $k_{\max} = 3$. The fourth column presents the size of the compressed file using the original PPM algorithm. The compression performance of our proposed method PPM+F is shown in the last two columns, where the net size of the encoding is depicted in the fifth column and the total size, including the header, whose size is bounded by $O(|\Sigma| \log n)$, is given in the last column, headed *net-encoding* and *total*, respectively.

On the shown examples the net encoding for PPM+F slightly improves on PPM. For *ftxt* and *XML*, while the compressed file itself is smaller for PPM+F than for PPM alone, the addition of the header increases 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.

We wish to emphasize that the purpose of the chosen experiments was not to show significant improvements by our newly suggested method. The difference between the classical PPM and the new method combining it with some features of the Forward looking paradigm relates only to the first appearances of each of the characters (context $k = -1$ in PPM), generally quite close to the beginning of a file, as well as to their disappearance towards the end of the files. We therefore expect the corresponding encoding sizes to differ only at the beginning and ending of the compressed files, and these differences will become less significant for increasingly larger test files. On the other hand, restricting the tests only to very small files would not allow us to show the full power of PPM, whose good performance depends on the availability of significant statistics for ever-growing contexts.

Conclusion

In this paper, we integrate the Forward Looking variant of dynamic Huffman or arithmetic encoding with the PPM algorithm, and provide empirical evidence for the improvement of the compression efficiency. The main idea of the combined algorithm is to predict the following character based on the symbols that have already been processed as well as using the knowledge of the exact frequencies in the text that are still to come.

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 minor one, for certain types of files.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Cleary JG, Witten IH. Data compression using adaptive coding and partial string matching. IEEE Trans Commun. 1984;32(4):396–402.
2. Huffman DA. A method for the construction of minimum-redundancy codes. Proc IRE. 1952;40(9):1098–101.
3. Witten IH, Neal RM, Cleary JG. Arithmetic coding for data compression. Commun ACM. 1987;30(6):520–40.
4. Elias P. Universal codeword sets and representations of the integers. IEEE Trans Inf Theory. 1975;21(2):194–203.
5. Klein ST, Shapira D. Random access to Fibonacci encoded files. Discrete Appl Math. 2016;212:115–28.
6. Tunstall BP. Synthesis of noiseless compression codes. PhD thesis, Georgia Institute of Technology. 1967.
7. Klein ST, Shapira D. On improving Tunstall codes. Inf Process Manage. 2011;47(5):777–85.
8. Ziv J, Lempel A. A universal algorithm for sequential data compression. IEEE Trans Inf Theory. 1977;23(3):337–43.
9. Ziv J, Lempel A. Compression of individual sequences via variable-rate coding. IEEE Trans Inf Theory. 1978;24(5):530–6.
10. Storer JA, Szymanski TG. Data compression via textual substitution. J ACM. 1982;29(4):928–51.
11. Welch TA. A technique for high-performance data compression. IEEE Comput. 1984;17(6):8–19.
12. Moffat A, Turpin A. Compression and Coding Algorithms. In: The international series in engineering and computer science, 1st ed., vol. 669. Kluwer, Springer, English; 2002
13. Klein ST, Saadia S, Shapira D. Forward looking Huffman coding. Theory Comput Syst. 2021;65(3):593–612.
14. Fruchtman A, Klein ST, Shapira D. Bidirectional adaptive compression. In: Proceedings of the Prague Stringology Conference 2019; 2019. pp. 92–101.
15. Klein ST, Shapira D. A new compression method for compressed matching. In: Data compression conference, DCC 2000, Snowbird; 2000. pp. 400–409.
16. Vitter JS. Algorithm 673: Dynamic Huffman coding. ACM Trans Math Softw. 1989;15(2):158–67.
17. Fruchtman A, Gross Y, Klein ST, Shapira D. Weighted adaptive coding. CoRR **abs/2005.08232**; 2020.
18. Fruchtman A, Gross Y, Klein ST, Shapira D. Backward weighted coding. In: Data compression conference, DCC 2000, Snowbird; 2021. pp. 93–102
19. Fruchtman A, Gross Y, Klein ST, Shapira D. Weighted Burrows-Wheeler compression. CoRR **abs/2105.10327**; 2021.
20. Véronis J, Langlais P. Evaluation of parallel text alignment systems: the ARCADE project. In: Véronis J, editor. Parallel text processing, Chapter 19. Dordrecht: Kluwer Academic Publishers; 2000. pp. 369–388.