PPM with extended alphabet

Przemysław Skibiński
Institute of Computer Science, University of Wrocław,
Przesmyckiego 20, 51-151 Wrocław, Poland,
inikep@ii.uni.wroc.pl

October 21, 2003

This is a preprint of an article published in
Information Sciences, 2006; 176(7):861–874
Copyright ©2006 Elsevier Inc.
Available online at www.sciencedirect.com

Abstract

In the following paper we propose modification of Prediction by Partial Matching (PPM)—a lossless data compression algorithm, which extends an alphabet, used in the PPM method, to long repeated strings. Usually the PPM algorithm's alphabet consists of 256 characters only. We show, on the basis of the Calgary corpus [1], that for ordinary files such a modification improves the compression performance in lower, but not greater than 10, orders. However, for some kind of files, this modification gives much better compression performance than any known lossless data compression algorithm.

KEY WORDS: lossless data compression, PPM, Prediction by Partial, repeated strings

1 Introduction

Prediction by Partial Matching [2] (PPM), with its many variations (PPMD [3], PPMZ [4], PPMII [5]), has been, for many years, the best lossless data compression algorithm, if we look at the compression ratio. This can be observed in *Table 1.* In 2002, Shkarin proposed a variation of the PPM algorithm—PPM with Information Inheritance (PPMII [5]). This algorithm sets a new standard on the compression performance. The implementation of the PPM algorithm—PPMd [6], created also by Shkarin, showed that better compression performance could be achieved with similar memory and processor requirements like for widespread algorithms LZ77 [7] and BWT [8].

The following study is based on a work of Bentley and McIlroy [9], which presents the idea of finding long repeated strings in a file. The algorithm, demonstrated here, is not limited to text files, since neither the PPM algorithm nor the algorithm proposed by Bentley and McIlroy are limited to text files.

This paper is divided into several parts. First, the author describes the PPM algorithm itself. Then we present the idea of the extended alphabet. The next chapter presents the results of the experiments on the author's implementation of the PPM algorithm with the extended alphabet. At finally, there are conclusions and bibliography.

Table 1: History of the best compression results on the Calgary corpus. The results are in bits per character.

File	pack[1]	Compress[2] 1985	PPMC[3] 1988	PPMD[4] 1993	PPMD+[5] 1995	PPM-FSMX[6] 1997	CTW+PPM*[7] 1998	cPPMII[8] 2002	PAQ[9] 2003
bib	5.23	3.35	2.11	1.88	1.862	1.786	1.714	1.663	1.617
book1	4.56	3.46	2.48	2.31	2.303	2.184	2.150	2.119	2.090
book2	4.83	3.28	2.26	1.97	1.963	1.862	1.820	1.742	1.691
geo	5.69	6.08	4.78	4.74	4.733	4.458	4.526	3.869	3.536
news	5.23	3.86	2.65	2.38	2.355	2.285	2.210	2.084	2.034
obj1	6.08	5.23	3.76	3.74	3.728	3.678	3.607	3.345	3.047
obj2	6.30	4.17	2.69	2.42	2.378	2.283	2.245	1.898	1.703
paper1	5.03	3.77	2.48	2.34	2.330	2.250	2.152	2.122	2.052
paper2	4.65	3.52	2.45	2.32	2.315	2.213	2.136	2.112	2.056
pic	1.66	0.97	1.09	0.80	0.795	0.781	0.764	0.693	0.456
progc	5.26	3.87	2.49	2.38	2.363	2.291	2.195	2.106	2.031
progl	4.81	3.03	1.90	1.70	1.677	1.545	1.482	1.352	1.314
progp	4.91	3.11	1.84	1.71	1.696	1.531	1.460	1.360	1.312
trans	5.58	3.27	1.77	1.50	1.467	1.352	1.256	1.151	1.126
average	4.99	3.64	2.48	2.30	2.283	2.177	2.123	1.972	1.861

2 The PPM algorithm

Prediction by Partial Matching (PPM) is a compression algorithm, originally developed by Cleary and Witten [2]. PPM is an adaptive, statistical compression method. A statistical model accumulates count of symbols (usually characters) seen so far in the input data. Thanks to that, an encoder can predict probability distribution for new symbols from the input data. Then a new symbol is encoded with a predicted probability by an arithmetic encoder. If the probability is higher, the arithmetic encoder needs fewer bits to encode the symbol and the compression performance is better.

The statistical PPM model is based on *contexts*. The context is a finite sequence of symbols preceding the current symbol. The length of the sequence is called *order of the context*. The *context model* keeps information about count of symbols' appearances for the context. All context models build the PPM model. Maximal order of the context is called *PPM model's order* or shorter *PPM order*. The contexts are built adaptively from scratch. They are created during the compression process. While encoding a new symbol from the input data, we are in some context, called *active*. Order of this context is between 0 and n, where n is PPM model's order. After encoding the new symbol, the PPM model is updated. The counters of the symbol for contexts models from active order until n are updated. This technique is called *update exclusion*. If some contexts do not exist, they are created. Updating contexts of lower order may distort probabilities, which may results in worse compression performance. The PPM model has a special order -1, which contains all symbols from the alphabet with equal probability.

Several factors influence settlement of a PPM model's order. Higher order is associated with larger memory requirements but also with better estimation of a probability, that is also with better

[1]pack (Huffman) — results taken from Reference [11]
[2]compress (LZW) — results taken from Reference [11]
[3]PPMC — results taken from Reference [**?**]
[4]PPMD — experimental results taken from the author's implementation of PPMD
[5]PPMD+ — results taken from Reference [13]
[6]PPM-FSMX — results taken from Reference [14]
[7]CTW+PPM* — results taken from Reference [15]
[8]cPPMII — experimental results taken by the author from PPMonstr var. I [6]
[9]PAQ — experimental results taken by the author from PAQ6 [**?**]

compression performance. Sometimes a symbol, which we have to encode, has not appeared in the context and the probability for this symbol is equal to zero. It happens especially in higher orders. Therefore, we add, to each context, a new symbol called *escape*, which switches the PPM model to a shorter context. The estimation of probability for the escape symbol is a very important and difficult task. There are many varying methods of selection of the escape symbol's probability: PPMA [2], PPMB [2], PPMC [10], PPMD [3], PPMZ [4] and PPMII [5], to name the most important ones only. As the higher order causes deterioration in the compression performance, the most often applied order for widely used PPMD is five. This is caused by frequent occurrence of the escape symbol and its bad estimation. However, estimation of the escape symbol's frequency for PPMII is much better. PPMII uses orders even up to 64, but the main reason allowing to use such high orders is much better estimation of ordinary symbols' probability (thanks to using an auxilliary model together with the standard PPM model). On the other hand, high orders are in practice rarely used as they have much higher memory requirements.

We have assumptions for PPMA, PPMB, PPMC, and PPMD about sources' escape frequency. For example, escape frequency in PPMD for the context c is equal $(u/2)/n$, where u is the number of unique symbols seen so far in the context, and n is the number of all symbols seen so far in the context. Escape estimation in PPMII and PPMZ is adaptive. It uses a secondary escape model (SEE [4]). SEE is a special, separate model used for better evaluation of probability for escape symbols only. Most PPM models use statistics from the longest matching context. PPMII inherits the statistics of shorter contexts when a longer context is encountered for the first time. The shorter (the last longest matching) context's statistics are used to estimate the statistics of the longer context.

3 The extended alphabet

3.1 Basic ideas

Let us assume that we have a file, in which we can find a long repeated (at a random distance) string with the length of s bytes (characters). PPM has to encode each character of this string separately (but with very high probability). PPM must always have a possibility to encode the escape symbol (when a new symbol in context has appeared). In this case, it is probable that the LZ77 algorithm would compress better, as it encodes only the offset, the length, and the next character. But if we extend the PPM alphabet with a new symbol, which is equal to our string (with the length s), we can encode this string much more efficiently. Let us assume that a dictionary is set of all long repeated strings. In this case, our alphabet consists of 256 characters (the ordinary PPM alphabet), the escape symbol, and symbols, which are equal to long repeated strings from the dictionary. We have to pre-pass over data to calculate statistics and find repeated strings to construct the alphabet before we start the PPM compression.

3.2 Related work

Using other alphabet than 256 characters in the PPM algorithm is not a new idea. Moffat used it in the Word algorithm [16]. The Word algorithm is similar to the PPM algorithms. The original text is divided into sequence of *words* and *non-words*. The Word algorithm uses an alphabet of words–comprising letters and non-words—comprising other characters. Teahan and Cleary [17] developed the PPM compression based on words and tagged words (modelled by parts of speech). However, these two approaches are limited to text files and also are not competitive to recent character-based PPM variations.

It seems that the idea introduced in this paper is similar to the idea of connection of data compression with algorithms PPM and LZ77 using the switching method presented in Reference [18]. These two approaches differ considerably, since PPM+LZ77:

- is limited by the sliding window, while the algorithm, introduced here, does not depend on the distance between the appearances of identical strings,

- requires encoding of switches between algorithms PPM and LZ77, while the algorithm, demonstrated in the following paper, does not require switches,
- needs to encode offsets and its lengths, while the algorithm, introduced here, encodes symbols from the alphabet only (and its lengths, but only during the first appearances),

These points are advantages of the introduced PPM algorithm modification over PPM+LZ77, but there are also disadvantages, since our modification:

- has to scan the whole data before starting compression to build the dictionary and this eliminates a feature of PPM – streaming processing,
- sets on us limitations as we use only the symbols from the dictionary.

If we create too large dictionary, instead of getting better compression performance, we can deteriorate it. Therefore, selection of strings, which are building the dictionary, is very important.

3.3 Finding long repeated strings

In the paper of Bentley and McIlroy [9], we can find description of a very good algorithm, which has been created to find long repeated strings. This is a pre-compression algorithm that can interact with many known compression algorithms. It is based on the Karp–Rabin's algorithm [19] to find pattern in a text (with using of a fingerprint). Data are divided into parts with the length b. Each part has its own fingerprint. Fingerprints are stored in a hash table. In the following step, we are scanning the data, computing the fingerprints for each string of the length b. When the string's fingerprint can be found in the hash table, we have a very high chance that we found the repeated string. If the strings are equal, we can extend them to all common characters before and after these strings.

The Karp and Rabin's algorithm can find all the occurrences of a string s of the length m in the input text of the length n. A *fingerprint* is made from m characters of the string s as a polynomial modulo of a large prime number. The Karp and Rabin's algorithm scans the input text, computing the fingerprint for each of $n - m + 1$ substrings of the length m. If the fingerprint of the string s and the fingerprint of a substring are not equal, we are certain that the substring does not match the string s. When fingerprints are equal, we have to check if the string s does match the substring. Karp and Rabin proved that a fingerprint can be computed in $O(m)$ time and updated, while scanning an input text, in $O(1)$ time. The worst-case complexity of this algorithm is $O(m \cdot n))$. For ordinary data, however, it runs close to linear time.

The Bentley and McIlroy's algorithm divides the input data into blocks of the size b. For an each block a fingerprint is created to give n/b fingerprints. The fingerprints are stored in a hash table. When we scan input data and evaluate the fingerprints, we can find equal fingerprints in the hash table. When fingerprints do match and blocks of the size b do match, we can extend these blocks to all common blocks and characters before and after the blocks. The Bentley and McIlroy's algorithm detects all repeated strings with the length at least $2b-1$, may detect repeated strings with the length from b through $2b-2$, and will not detect repeated strings with the length less than b. If the strings do overlap, it must be cut (shortened) not to overlap.

Bentley and McIlroy claim that the worst-case complexity of this algorithm is $\Theta(n^{3/2})$, but for "realistic" data, it runs close to linear time. This algorithm is constructed especially for long repeated strings and nowadays it is the fastest known algorithm solving this problem. In the next subsection one is given a description of how to select value of b.

3.4 Which repeated strings should be added to the dictionary

The alphabet expansion can slightly worsen the compression performance. By choosing too short repeated strings, we can also deteriorate the compression performance.

It is very important how we choose the dictionary and the author suggests the following gain function:

$$gain(s) = |s| \cdot (count(x) - 1).$$

Where $|s|$ is the length of the string s, $count(x)$ is the count of strings in compressed data. This is a very simple function that describes how many characters from input will be replaced by symbols from the dictionary. We may say that this is the amount of bytes that are saved. We do not take into consideration the escape symbols (because usually to use a symbol from the dictionary, we have to switch to the order -1) and the fact that a replaced string would be compressed with the ordinary PPM alphabet.

Having the gain function, we can add to the alphabet only the strings s, for which $gain(s) \geq y$, where y is constant, different for each file (even for a different compression algorithm, e.g., PPMD and PPMII). Now we can evaluate value of b from the Bentley and McIlroy's algorithm. In order to find all the repeated strings with the length $|s|$, we select $|s| = 2b - 1$ and we get $b = (|s| + 1)/2$.

3.5 Efficient storage of the dictionary in the compressed data

At the first appearance of a string we encode the string using the standard PPM alphabet. After string encoding, we encode its symbol (from the extended alphabet) and the length of the string. Thanks to that, we can evaluate the string's start position. We also know the length of the string and we have the unique symbol that is equal to this string. While the next appearances of the same string, we encode only the symbol of this string (which will be replaced with this string while decoding). We do not need any additional information. There are only two variants of symbol appearance—the first appearance of the symbol followed by the length of the string and the next appearances of the symbol without the length of the string.

3.6 Memory requirements

The Karp-Rabin's algorithm splits a file into blocks of the size b. For each block we need four bytes for the fingerprint and four bytes for the block position (or the number). We can estimate the memory required for the Karp-Rabin's algorithm (without the size of the created dictionary) with the equation:

$$mem = n + 8 \cdot \frac{n}{b}.$$

When we select $b = (|s| + 1)/2$ then we get the equation:

$$mem = n + 16 \cdot \frac{n}{|s| + 1},$$

where mem is the required memory, n is the length of the file and $|s|$ is the minimal length of the repeated string s.

When the file size is very high, we can split the file into parts (of the size, e.g., 5 MB) and find repeated strings for each part. In the following step we can join the same strings from different parts of the file. If we split the file into the parts, the compression performance may be decreased, but in the majority of cases only slightly. If we know the amount of memory (allocated buffer) that will be used by the PPM algorithm, we can create the dictionary using the same memory. It can be done by splitting the file into the parts that the required memory will be less than size of the allocated buffer.

4 Results

4.1 Experiments on the Calgary Corpus in order 1

The Calgary Corpus has been, for many years, the most known set of files created for testing lossless data compression algorithms. The results of experiments are given in *bits per character*

Table 2: Comparison of PPMD-1 and PPMD-1 with the extended alphabet ($|s| \geq 5$, $gain(s) \geq 30$)

File	PPMD order 1	PPMD order 1 + dictionary	Dictionary (new symbols in alphabet)	Count of new symbols
bib	3.461	2.416	508	9828
book1	3.605	2.952	3252	87553
book2	3.768	2.612	2284	70148
geo	4.660	4.542	14	417
news	4.156	3.086	1901	29617
obj1	4.517	4.266	74	673
obj2	4.065	3.287	1065	20399
paper1	3.807	3.027	330	4409
paper2	3.623	2.919	468	7729
pic	0.840	0.882	547	13763
progc	3.827	3.064	234	3252
progl	3.306	2.254	359	6933
progp	3.342	2.380	243	4687
trans	3.477	2.029	381	6308
average	3.604	2.837		

(bpc). To get results in bpc, we calculate the equation $result = (8 \cdot ofs)/ifs$, where *ofs* is the output file size and *ifs* is the input file size. Lower bpc means, of course, better compression performance.

At first, the author carried out experiments with PPMD[10], order 1, with the standard alphabet (256 characters). Then the author tested the same algorithm with added dictionary, created with assumptions $|s| \geq 5$, $gain(s) \geq 30$. These values are chosen experimentally, as optimal value for each file is different. *Table 2* contains information about how many new symbols were added into the alphabet as well as sums of appearances for these symbols.

We can see that the proposed modification leads to much better compression efficiency for all the files except *pic*. This is caused by the fact that long repeated strings of the same characters (e.g., "xxxxxxx...") are compressed very well by the ordinary PPM algorithm. In this case our modification worsens the compression performance. But even in such a low order, measured result 2.837 bpc is comparable with algorithms from LZ77 family (ZIP, gzip [20]).

4.2 Experiments on the Calgary Corpus in order 5

In the following step, the author carried out experiments with the proposed algorithm on PPMD, order 5, and PPMII [11], order 5. The results can be seen in *Table 3*. The dictionary and the count of new symbols for PPMD-5 with the extended alphabet and PPMII-5 with the extended alphabet are the same, because these are evaluated before starting of the PPM compression.

Alphabet extensions, in order 5, affect much only the three last files (*progl*, *progp*, *trans*). For files *bib*, *book2*, *geo*, *news*, *obj2*, *paper2*, and *progc*, we have insignificant improvement of the compression performance for both PPMD and PPMII. This time we can see improvement of compression performance for the file *pic* as a considerably smaller dictionary was created for the parameters $|s| \geq 30$, $gain(s) \geq 200$. This means that there are fewer (or not at all) long repeated strings of the same characters in the dictionary. For the file *book1* there is no change of the compression performance.

For files *obj1* and *paper1* insignificant deterioration of the compression performance occurs, which happens because the chosen parameters $|s|$ and $gain(s)$ are inappropriate for these files. The gain function $gain(s)$ and $|s|$ was chosen for good compression performance for the last three

[10] PPMD — the author's implementation of the PPMD algorithm described in [3]

[11] PPMII — the author's implementation of the PPMII algorithm described in [5], that has a little worse compression performance than PPMd by Shkarin

Table 3: Comparison of PPM-5 and PPM-5 with the extended alphabet ($|s| \geq 30$, $gain(s) \geq 200$)

File	PPMD order 5	PPMD order 5 + dictionary	PPMII order 5	PPMII order 5 + dictionary	Dictionary (new symbols in alphabet)	Count of new symbols
bib	1.876	1.860	1.785	**1.772**	28	213
book1	2.275	2.275	**2.214**	**2.214**	1	2
book2	1.952	1.949	1.898	**1.895**	27	248
geo	4.832	4.829	4.510	**4.506**	3	45
news	2.368	2.343	2.280	**2.259**	100	775
obj1	3.785	3.791	**3.656**	3.663	5	17
obj2	2.429	2.393	2.337	**2.305**	85	505
paper1	2.335	2.336	**2.238**	2.239	2	7
paper2	2.303	2.301	2.210	**2.209**	2	7
pic	0.807	0.811	0.792	**0.786**	113	5488
progc	2.385	2.382	2.268	**2.266**	2	5
progl	1.682	1.630	1.603	**1.557**	34	182
progp	1.717	1.638	1.613	**1.547**	30	71
trans	1.496	1.420	1.381	**1.331**	77	352
average	2.303	2.283	2.199	2.182		

files from the Corpus. If we choose $gain(s)$ and $|s|$ different for each file, we can improve the compression performance even by about 0.01 bpc.

The most interesting file is *pic*. Using the extended alphabet, on the one hand we get deterioration of the compression performance on PPMD and on the other hand improvement of the compression performance on PPMII. This is caused by the differences between PPMD and PPMII algorithms themselves. The exact reason is that coding a symbol from the extended alphabet requires in many cases switching to very low order (e.g., to the order -1 during the first appearance of the symbol) in which PPMII better estimates probabilities of appearance of the escapes. The file *pic* is very specific and PPMII shows the superiority in the compression performance over PPMD on this file.

Average compression performance is improved for about 0.02 bpc for both PPMD and PPMII. In higher orders differences are even lower. In orders higher than 10 there is no gain at all. The files from the Calgary Corpus are too small to get better compression performance from the introduced modification.

4.3 Experiments on the Large Cantenbury Corpus

The Large Cantenbury Corpus [21] is a collection of relatively large files. PPM with extended alphabet requires very large amounts of data to get good compression performance. On this corpus we can observe how our modification manages with a large files (see *Table 4*). The experiments are carried out with varying parameters $|s|$ and $gain(s)$.

For the same parameters $|s|$ and $gain(s)$ on the one file we have deterioration of the compression performance, on the other we have improvement of the compression performance. It depends on the content of a file itself. There are the two conclusions from the experiments.

First, when we use values $|s| \geq 25$ and $gain(s) \geq 250$ our modification should in the majority of cases improve the compression performance. The results with values $|s| \geq 30$, $gain(s) \geq 200$ on the Large Cantenbury Corpus are close. Similar results can be seen in the earlier experiments.

Second. The results on the large files with values $|s| \geq 5$, $gain(s) \geq 300$ are better than expected. There is the deterioration of the compression performance on the file *E.coli* when $|s| \geq 10$, but this file is very specific. It contains only the letters *a*, *g*, *c*, *t*. For such a small alphabet is trivial to compress the file to 2 bpc, but is very hard to compress better than 2 bpc.

Table 4: Comparison of PPMII-5 and PPMII-5 with the extended alphabet with varying parameters $|s|$ and $gain(s)$

File	PPMII	PPMII + dictionary					
		$\|s\| \geq 25$, $gain \geq 250$	$\|s\| \geq 15$, $gain \geq 250$	$\|s\| \geq 10$, $gain \geq 250$	$\|s\| \geq 7$, $gain \geq 300$	$\|s\| \geq 5$, $gain \geq 300$	$gain_{clx} \geq 0$
bible.txt	1.564	1.551	1.555	1.570	1.563	**1.487**	1.545
E.coli	1.944	1.920	1.920	1.945	2.131	2.061	**1.918**
world192.txt	1.459	1.364	1.338	1.333	1.316	**1.306**	1.338
average	1.656	1.612	1.604	1.616	1.670	1.618	**1.600**

There is too many repeated strings in the extended alphabet when $|s| \geq 10$ as the strings are better compressed with the ordinary PPM compression.

For the files *bible.txt* and *world192.txt* we have significant improvement of the compression performance, 0.077 and 0.153, respectively. These results are similar to the standard PPMII algorithm's results in order 8. About half of the file *world192.txt* was compressed using extended alphabet. The main problem is that for $|s| \geq 5$ the proposed pre-compression algorithm is several times slower than for $|s| \geq 25$. Also its memory requirements are about four times higher.

4.4 Complex gain function

Although the gain function is evaluated in a very simple way, it gives quite good results. We can try to estimate real gain from adding a string to the extended alphabet to get better compression performance. The experiments on the Large Cantenbury Corpus leaded the author to develop new gain function. This function is designed to be parameterless. Lets look on our gain function:

$$gain(s) = |s| \cdot (count(x) - 1).$$

Where $|s|$ is the length of the string s, $count(x)$ is the count of strings in compressed data. We have two parameters using while the compression, y and z, in inequalities $|s| \geq y$ and $gain(s) \geq z$. As we remember from earlier experiments, these parameters depend on maximum PPM order. We can set these parameters to the constants: $|s| \geq 2$ and $gain(s) \geq 0$. This should worsen the compression for our gain function, but the author has developed the new, complex gain function:

$$gain_{clx}(s) = \frac{5 \cdot |s| \cdot (count(x) - 1)}{order_{max}} - 2 \cdot order_{max} \cdot (count(x) - 1) + 8 \cdot order_{max} - 120$$

This function is parameterless and depends also on $order_{max}$, which is maximum PPM order.

Complex gain function's results on the Large Cantenbury Corpus can be seen in *Table 4*. The complex gain function gives the best results in average and the best result on the file *E.coli* (which is one of the best published results on this file). As we were expected, the gain from the introduced modification on the files from the Large Cantenbury Corpus is higher (about 0.06 bpc) than on the files from the Calgary Corpus (about 0.02 bpc).

4.5 Experiments on the file bible.txt (from the Large Cantenbury Corpus)

Table 5 contains the results of the experiment similar to the experiment described by Bentley and McIlroy in Reference [9]. It confirms that PPMII compresses much better than gzip. For the file *bible.txt*, the alphabet extension gives better compression performance (what we have already seen in earlier experiments). In higher orders PPMII has better compression performance and the alphabet extensions gives less gain. Multiple joining (concatenation) of the same file and, later on, compression with PPMII with the extended alphabet gives the large gain.

Table 5: Comparison of gzip, PPMII and PPMII with the extended alphabet ($gain_{clx} \geq 0$). Results are given in bytes.

File	uncompressed	gzip	PPMII-5	PPMII-5 + dictionary	PPMII-8	PPMII-8 + dictionary
bible	4047392	1191907	791448	781523	741096	739174
bible+bible	8094784	2383216	1513502	781531	1258525	739183
bible+bible+bible+bible	16189568	4765814	2923811	781537	2213626	739190

Gzip has a small sliding dictionary window (32KB) and almost does not use similarities between distant strings at all. PPMII-5 uses similarity in strings, but order 5 is too low for good probability estimation for the file *bible.txt*. That causes worse compression performance than expected. PPMII-8 better estimates probabilities and gives about 2 times smaller output file than gzip on the concatenated file *bible.txt*. PPMII with the extended alphabet compresses almost without difference, no matter how many times we have concatenated the file. This is caused by the fact that the whole file *bible.txt* is treated like a single symbol from the extended alphabet.

5 Conclusions

In this paper the author presented a universal modification of any algorithm from the PPM family. Low orders in the PPM algorithm are often used to limit memory requirements. In our modification better compression performance can be achieved in the same memory requirements. Compression and decompression speed for the ordinary PPM algorithm is almost identical, since the encoder and the decoder perform similar tasks. The proposed modification decreases compression speed, as we have to find long repeated strings before starting of the PPM compression. Decompression speed is higher because strings from the extended alphabet are copied from the dictionary and there is no need to modify a context tree for each character from this string. This approach is especially desirable for the off-line compression [11, 22, 23, 24], where only time of decompression is significant.

The method, introduced in this paper, can be improved to get even better compression performance. It can be done by introduction of a new symbol, namely the *escape/switch* symbol, without removing the escape symbol from the alphabet. The gain from using the extended alphabet in high orders is partly eliminated with necessity to apply series of escapes to switch into the order -1. Usually only in this order we can find symbols from the extended alphabet. By applying the additional symbol escape/switch, which appears in each context, we can switch instantly to the order -1 or even to the extended alphabet itself. The escape/switch symbol, however, must have a very small probability not to worsen the compression performance, instead of improving.

The author has shown that the extension of the PPM alphabet gives better compression performance in orders not greater than 10. For specific kind of files, that is for the files with long repeated strings (e.g., concatenation a file with itself), the algorithm is capable of reaching much better compression performance than any known lossless data compression algorithm.

Acknowledgements

The author wishes to thank Sebastian Deorowicz and Szymon Grabowski for great help in improving this document as well as Uwe Herklotz, Dmitry Shkarin and Maxim Smirnov for helpful comments and advices.

References

[1] Bell, T.C., Cleary, J., Witten, I.H. (1990) *Text compression. Advanced Reference Series.* Prentice Hall, Englewood Cliffs, New Jersey. `http://corpus.canterbury.ac.nz/descriptions/`

#calgary

[2] Cleary, J.G., Witten, I.H. (1984) Data compression using adaptive coding and partial string matching. *IEEE Trans. Commun.*, **COM-32** (4), 396–402.

[3] Howard, P.G. (1993) The design and analysis of efficient lossless data compression systems. *Technical Report CS–93–28*, Brown University, Providence, Rhode Island. `ftp://ftp.cs.brown.edu/pub/techreports/93/cs93-28.ps.Z`

[4] Bloom, C. (1998) Solving the problems of context modeling. `http://www.cbloom.com/papers/ppmz.zip`

[5] Shkarin, D. (2002) PPM: one step to practicality. In *Proc. Data Compression Conf.*, pp. 202–211. `http://datacompression.info/Miscellaneous/PPMII_DCC02.pdf`

[6] Shkarin, D. (2002) PPMd—fast PPMII compressor. `http://compression.ru/ds`, `http://www.simtel.net/pub/pd/18902.html`

[7] Ziv, J., Lempel, A. (1977) A universal algorithm for sequential data compression. In *Proc. IEEE Trans. Inform. Theory*, **IT-23** (3), 337–343.

[8] Burrows, M., Wheeler, D.J. (1994) A block-sorting lossless data compression algorithm. *SRC Research Report 124*, Digital Equipment Corporation, Palo Alto, California. `ftp://ftp.digital.com/pub/DEC/SRC/research-reports/SRC-124.ps.zip`

[9] Bentley, J., McIlroy, D. (2001) Data compression with long repeated strings. *Information Sciences* **135** (1–2), 1–11.

[10] Moffat, A. (1990) Implementing the PPM data compression scheme. *IEEE Trans. Commun.*, **COM-38** (11), 1917–1921.

[11] Apostolico, A., Lonardi, S. (2000) Off-line compression by greedy textual substitution. In *Proc. Data Compression Conf.*, pp. 143–152.

[12] Teahan, W.J. (1995) Probability estimation for PPM. In *Proc. 2nd New Zealand Computer Science Research Students' Conference*, University of Waikato, Hamilton, New Zealand. `ftp://ftp.cs.waikato.ac.nz/pub/papers/ppm/pe_for_ppm.ps.gz`

[13] Bloom, C. (1996) LZP—a new data compression algorithm. In *Proc. Data Compression Conf.*, pp. 1996. `http://www.cbloom.com/papers/lzp.zip`

[14] Bunton, S. (1997) Semantically motivated improvements for PPM variants. *Comp. J.* **40** (2-3), 76–93.

[15] Volf, P.A.J., Willems, F.M.J. (1998) The switching method—elaborations. In *Proc. 19th Symposium on Information Theory in the Benelux*, pp. 13–20.

[16] Moffat, A. (1989) Word based text compression. *Software—Practice and Experience*, **19** (2), 185–198.

[17] Teahan, W.J., Cleary, J.G. (1998) Tag based models of English text. In *Proc. Data Compression Conf.*, pp. 43–52.

[18] Volf, P.A.J., Willems, F.M.J. (1998) Switching between two universal source coding algorithms. In *Proc. Data Compression Conf.*, pp. 491–500.

[19] Karp, R.M., Rabin, M.O. (1987) Efficient randomized pattern-matching algorithms. *IBM J. Res. Develop.*, **32** (2), 249–260.

[20] Gailly, J.-L. (1993) GNU zip documentation and sources. `http://www.gzip.org`

[21] Ross, A., Bell, T.C. (1997) A corpus for the evaluation of lossless compression algorithms. In *Proc. Data Compression Conf.*, pp. 201–210. `http://corpus.canterbury.ac.nz/descriptions/#large`

[22] Apostolico, A., Lonardi, S. (1998) Some theory and practice of greedy off-line textual substitution. In *Proc. Data Compression Conf.*, pp. 119–128.

[23] Larsson, N.J., Moffat, A. (1999) *Off-line dictionary-based compression.* In *Proc. Data Compression Conf.*, pp. 296–305.

[24] Turpin, A., Smyth, W.F. (2002) An approach to phrase selection for off-line data compression. In *Proc. 25th Australasian Conference on Computer Science*, Melbourne, Australia, **Volume 4**, pp. 267–273.

[25] Smith, T.C., Peeters, R. (1998) Fast convergence with a greedy tag-phrase dictionary. In *Proc. Data Compression Conf.*, pp. 33–42.