

Variable-length contexts for PPM

Przemysław Skibiński¹ and Szymon Grabowski²

¹ Institute of Computer Science, University of Wrocław, Wrocław, Poland,
e-mail: inikep@ii.uni.wroc.pl

² Computer Engineering Department, Technical University of Łódź, Łódź, Poland
e-mail: sgrabow@zly.kis.p.lodz.pl

This is a preprint of an article presented at DCC'04, published in
Proceedings of the IEEE Data Compression Conference (DCC'04), page 409-418
Copyright ©2004 IEEE

Abstract

In this paper we present a PPM variation which combines traditional character based processing with string matching. Such an approach can effectively handle repetitive data and can be used with practically any algorithm from the PPM family. The algorithm, inspired by its predecessors, PPM* and PPMZ, searches for matching sequences in arbitrarily long, variable-length, deterministic contexts. The experimental results show that the proposed technique may be very useful, especially in combination with relatively low order (up to 8) models, where the compression gains are often significant and the additional memory requirements are moderate.

1 Introduction

Prediction by Partial Matching (PPM) is a well-established adaptive statistical lossless compression algorithm. Today its advanced variations offer best average compression ratios [7]. Unfortunately, the memory requirements for PPM are high and the compression speed relatively low. Generally, those drawbacks are becoming more acute with increasing the maximum context order. Higher orders provide more information necessary for reliable prediction but also increase the computational requirements. It should also be stressed that making effective use of high order information is a non-trivial task.

In this paper we present a modification of the state-of-the-art PPM algorithm, Shkarin's PPMII [15]. Our implementation is oriented towards repetitive data and equips the PPM with string matching capabilities. They alleviate the mentioned disadvantages of traditional PPM to some degree.

The paper is organized as follows. In Section 2 we present the background for our research. Section 3 describes our algorithm with several possible modifications. Section 4 discusses implementation details. The results of quite extensive experimental comparison tests are contained in Section 5.

2 Related work

One of the main problems with PPM compression is related to the high order contexts. They require large amounts of memory, lots of CPU power, and still are no magic remedy improving compression, since, due to scarcity of high-order statistics, exploiting them for more accurate prediction is difficult. Moreover, quite many data types do not seem to ever benefit from high order dependencies, so the best we could do for those files is just not to deteriorate compression in comparison with low-order modeling. Still, significant memory and time overhead is indispensable. Now the question is: when should long contexts be taken into account? Maybe from a practical point of view it is better to always stick to low/moderate order modeling? But then some kinds of data (e.g., redundant XML files) are compressed in a way far from optimal.

A logical and viable way of avoiding this dilemma seems to be using variable-length contexts. Several papers have inspired us for the research. We present the ideas in the chronological order.

2.1 PPM*: Contexts can be arbitrarily long

In most PPM schemes, the maximum context order is finite. Increasing the maximum order may lead to better compression, as higher order contexts are expected to be more specific in their predictions, but practitioners are well aware that this is not always the case. For example, on textual data most available PPM implementations achieve best results with the maximum order length set to five, and deteriorate slowly with increasing that value. It happens because high order contexts usually do not contain many symbols' occurrences, which implies that escapes to lower contexts are frequent, hence increasing the size of the encoded data.

On the other hand, it is obvious that for some data contexts much longer than (let us say) five are useful. Moreover, the "optimal" context lengths may vary within a file. For example, within a large collection of English texts the sequence "to be or not to" is very likely to be followed with symbol 'b', while truncating this context to order-5, i.e., to "t to" makes the prediction much more obscure.

Cleary et al. described the idea of using unbounded length contexts [5] in an algorithm named PPM*. This algorithm extends PPM with possibility of referring to arbitrarily long contexts (which may prove very useful on very redundant data) and is quite safe from a significant loss on data, which do not require such long contexts.

To explain how PPM* works, one simple definition is now needed. A context is called *deterministic* if only one symbol has appeared in it so far. There may be many occurrences of a given context, but as long as it predicts the same symbol, it remains deterministic. Cleary et al. noticed that in such contexts the probability of a novel symbol is much lower than expected based on uniform prior distribution. In other words, earlier PPM algorithms, handling all contexts in the same manner, had underestimated the prediction ability of deterministic contexts.

PPM* keeps a pointer to the input (uncompressed) data for each deterministic context. While encoding, the algorithm uses the shortest deterministic context from the list of available contexts (ancestors). By definition, the pointer to data always points onto a single symbol from the alphabet. If the prediction is correct, PPM* encodes the predicted symbol and moves the pointer forward. The sequence of successive correctly

predicted symbols can be arbitrarily long, therefore contexts have unbounded length. If the prediction fails, an escape is emitted. If there is no deterministic context for encoding a given symbol, the longest context is used instead. It means that in such cases PPM* behaves like an ordinary PPM coder.

2.2 PPMZ: Longer deterministic contexts are more trustworthy than shorter ones

PPMZ is a PPM variation developed by Bloom [2]. It uses several important techniques to boost compression: a heuristics for choosing the context order to start encoding from (LOE), an adaptive mechanism for escape probability estimation (SEE), and unbounded length deterministic contexts. The last technique is based on PPM*, but with some modifications. PPMZ uses only very long deterministic contexts (of length 12 or more) and imposes a minimum match length bound (for each context separately) to eliminate errors when the length of predicted symbols is too short.

The unbounded contexts in PPMZ are additional to the standard PPM model (in which the maximum order is less than 12) and in a way independent of it. The unbounded context model holds a pointer to the original (uncompressed) data and the minimum match length. PPMZ constructs unbounded context models adaptively during the encoding. They are built after encoding each symbol and indexed by a hashed value of the last 12 appeared symbols.

If an appropriate unbounded deterministic context is found (the same index value), PPMZ checks if its length is equal or greater than the minimum match length (i.e., 12). When an unbounded context is not found or the context length is too small, PPMZ uses the ordinary PPM model. If the length exceeds the minimum match length, PPMZ uses the pointer to encode symbols in the PPM* manner. Deterministic predictions are encoded until the first escape symbol. If the length of predicted symbols was too small, the minimum match length is changed to the context length plus one and the pointer is changed to point to the current data. It assures that this error will never occur again as similar unbounded contexts (reference and current, whose have the same index value) will be distinguished. PPMZ forces all unbounded contexts to be deterministic as only one context model can be referenced by the same index value.

2.3 PPMII: Your ancestors may teach you a lot

In 2002 Shkarin presented a paper on his PPM with Information Inheritance (PPMII) algorithm [15]. Most of the ideas described in the paper could earlier been known to the compression community from the public sources of his own implementation called (quite confusingly) PPMd [16]. The algorithm offers a very attractive trade-off between compression performance and speed, and thanks to that it had been chosen for implementation in several third-party archivers, including the well-known RAR [13]. Shkarin's PPM is a complex scheme and it is hard to describe it here in details, so we present below only its basic ideas.

The main novelty of PPMII (reflected in its name) is passing gathered information (statistics) from parent to child contexts. When a longer (child) context appears for the first time, it already contains some "knowledge" (probability estimation) inherited from its parent. In this way, the PPM model may learn faster and the problem with sparse statistics in high orders is mitigated.

Another significant idea is dividing the contexts into three classes to efficiently estimate escape probability and to correct probabilities of common symbols using separate models. Also important practical gains (although not fitting to the Markovian source model) are achieved via rescaling statistics and thus exploiting the recency (data non-stationarity). Together with some other minor improvements and very efficient data structures, PPMII can achieve very good compression ratios at the compression speed comparable to gzip (see PPMII, order-8, results on the Calgary Corpus in [15]). The main drawback of the scheme are quite significant memory requirements in high orders.

3 PPMVC: PPM with variable-length contexts

In this section we present a new technique based on the idea of unbounded length contexts – variable-length contexts (VC). As we have developed several versions of the algorithm, the section is divided into parts. We start from explaining the simplest version and later we extend it with new features.

3.1 PPMVC1 – the simplest and the fastest version

The PPM with variable-length contexts (PPMVC) algorithm virtually increases orders of maximum order contexts. One can say that PPMVC extends the traditional, character based PPM with string matching similar to the one used by the LZ77 algorithm.

The PPMVC mechanism uses PPM contexts of the maximum order only. If a context's order is lower, the VC technique is not used and the current symbol is encoded with traditional PPM model instead. The contexts do not have to be deterministic, unlike in PPM* or PPMZ. A pointer to the original data is kept in the context model. It is updated for each appearance of the context, no matter if the context was successful or not in its prediction. This assumption goes in accordance with Mahoney's experiments [11] intended to show that n -grams (i.e., contiguous sequences of n symbols) tend to reappear more often in close distances.

While encoding in the maximum order, PPMVC makes use of two pointers to the original data, one indicating the current position and the other a past position. The right match length (RML) is defined as the length of the matching sequence between those two data entries. The RML values have to be encoded explicitly, as they are not known by the decoder. The minimum length of the right match may be zero.

The PPMVC encoding is similar to PPMZ manner. While the encoder uses the maximum order context, it evaluates RML and encodes its value using an additional, global RML model (order-0). This is a significant difference to the coding variant from PPMZ; in our version there is no need to encode all correctly predicted symbols one by one (with a possibility of appearance of escape symbols).

There are two more ideas in PPMVC intended to improve compression. First is the minimum right match length (MinRML). If the current right match length is below the MinRML threshold, PPMVC sets RML to 0. This assures that short matches aren't used and they are encoded with a minimal loss. Second modification is dividing matched strings into groups and encoding only the group number. Groups are created by simply dividing RML by integer. For example, if the divisor is 3, the first group contains matched strings of length 0, 1 or 2, the second group contains strings of length 3, 4 or 5, and so on. In other words, the RML values are linearly quantized. Encoded RML must be bounded (e.g., the length of 255), but if the match length exceeds this bound, we can

encode the maximal RML and the remaining part as the next RML (possibly continuing this process).

3.2 PPMVC2 – added minimal left match length

The left match length (LML) is defined as the length of common sequence preceding the symbols pointed by the reference and the current pointer, respectively. As the pointed data are already encoded, both the encoder and the decoder can evaluate LML. The minimum length of the left match for these pointers is equal to the maximum PPM order, because this is the order of the reference and the current context that holds pointers.

PPMVC2 extends the previous version with a minimum left match length, which is based on a corresponding idea from PPMZ. For each context of maximum order, PPMVC2 remembers, besides the pointer to the original (uncompressed) data, the minimum match length.

While the encoder uses the context of maximum order, it evaluates LML using the reference and the current pointer. If LML is below the minimal left match length (MinLML) threshold defined for this context, the encoder uses traditional PPM. At first, the MinLML value is set to some constant, dependent on the maximum order (it will be explained later). If RML is zero, then MinLML for the context is updated with the formula $\text{MinLML} = \text{MinLML} + 1$. We can think about this formula as a very slow MinLML adaptation, when error occurs (RML is equal 0). Experiments have shown that for PPMVC this approach gives better results than Bloom's instant error-defending formula ($\text{MinLML} = \text{LML} + 1$ for PPMZ, where LML is the currently evaluated left match length).

3.3 PPMVC3 – the most complex version with the best compression

PPMVC2 remembers only last (most recent) pointer to original data for each context. The experiments have shown that this approach gives good results, but sometimes using a more distant reference can give better prediction (longer right match).

There are several possibilities of taking into account more than one pointer. One is to find the pointer with maximum RML and to encode RML. But in this case we must also explicitly encode the index (number) of the pointer to enable decompression. The pointer's index encoding must be effective enough not to deteriorate compression. Another possibility is to select a pointer which will be also known to the decoder. Good results should be achieved when we choose the pointer with the longest left match length (LML). We have implemented and compared these two alternatives.

The experiments have shown that in majority of cases the last (most recent) pointer gives the best prediction (longest right match). It suggests that indices can be efficiently encoded in the global index model (order 0), similar to the RML model. The last pointer has the index 0, the penultimate one index 1, and so on. The skew distribution of chosen pointers' indexes obviously implies their efficient encoding. The indexes are encoded adaptively by an index model. This approach decreases compression speed comparing to PPMVC2, but decompression speed is unaffected.

Index encoding is not necessary when we choose the index of the pointer with the longest LML. At first the encoder has to find the pointer with the longest LML. Then the encoder has to check if the match length exceeds MinLML. If so, the encoder can use this

pointer to encode the data, which is a reversible operation. Following this way, the compression and decompression speed is hampered in almost the same degree.

The main problem with the second approach (i.e., the longest LML) is that the longest LML not always predicts the longest right match. But it turns out that this idea gives better results than a variant in which the encoder would explicitly encode the index of the pointer with longer length of right match.

3.4 Selecting parameters

There are four parameters in PPMVC: the divisor d for match length quantization, MinRML, MinLML and the number of pointers per context (only for PPMVC3). These parameters mostly depend on the maximum PPM order (MO). When order is higher, PPM performs better in deterministic contexts (e.g., about 1 bpc in order 8 for textual data for PPMII), so the parameters d , MinLML and MinRML must respectively be greater.

Experiments with PPMII show that good average compression performance can be obtained if d is set to the MO+1. The right match length depends on the divisor d , and MinRML was set to $2d$. MinLML was chosen as $(MO-1) \cdot 2$.

The maximal number of the pointers depends also on the maximum PPM order. In higher orders contexts are better distinguished and there is no need to hold as many pointers as in lower orders. There are files for which the best compression ratios are obtained having from 32 to 64 pointers for order 4 (e.g., `nci` from Silesia Corpus [7]), but we suggest to hold 8 pointers, which results in best average compression performance. For order 8 we use 4 pointers, and only 2 pointers for order 16.

3.5 Summary

The memory requirements for the PPMVC algorithms are not very high. PPMVC1 needs only 4 bytes (for the pointer) for each context of maximum order. PPMVC2 uses 5 bytes (the pointer and 1 byte for MinLML value) here, while PPMVC3 requires $4n+1$ bytes (where n is the count of pointers).

At the end of this section, we would like to point out the main differences between PPMZ and PPMVC (PPMVC2 and PPMVC3):

- PPMVC simulates unbounded contexts using deterministic and non-deterministic contexts of maximum order, PPMZ creates additional contexts of constant (equal 12) order;
- PPMVC updates the pointer to the original data with last appeared context;
- different minimum match length update between PPMVC and PPMZ;
- PPMVC encodes only the length of well predicted symbols, PPMZ encodes all predicted symbols.

4 Implementation details

Our implementation of PPMVC [17] is based on PPMd [16]. PPMd seems to be the most successful file compressor currently available, at least among the open source software, reaching on average very good compression ratios at relatively high processing speed.

In our implementation of PPMVC3, pointers are held in a circular array of pointers for each context. The benefit of this solution is that memory for the array is allocated only once. In some contexts not all pointers may be used and some memory is wasted.

Much attention should be paid for the functions to compare strings (pointed current and reference data) and to find length of common sequence (left match length and right match length). These functions are invoked very often and they consume most of the CPU time, apart from the “ordinary” PPMII processing. We use a simple technique to increase speed, without affecting the compression performance. Having already found a string with locally maximal left match length (LMinLML), we look for the next match starting from symbol LMinLML down to symbol of index 0. The reversed direction is more efficient because almost all matches shorter than LMinLML are instantly eliminated, and, moreover, mismatches on more distant symbols are more probable than on close symbols. In other words, the bad matches will on average be eliminated faster.

To implement PPMVC, we have added two new files to PPMd sources, which comprise a file buffer handling and memory manager routines. The specifics of PPM with string matching (as opposed to pure symbol oriented PPM) require peeking into past data in their original representation. Here comes a question of compromise between memory use and compression effectiveness. Experiments showed that for large files providing a 4 MB buffer of most recent symbols leads to only slightly worse compression than using a buffer for the whole file. Another implementation aspect is a memory manager. For ease of use we chose a different manager than Shkarin’s one, which unfortunately implied increased overall memory use (the elementary unit size grew from 12 to 16 bytes). As a side-effect, it resulted in worse compression of large files (Silesia Corpus), especially in order-16, because of more frequent model overflows and restarts. In the future we are going to improve this aspect of our implementation.

5 Experimental results

All tests were run on an Athlon 1000 MHz machine equipped with 256 MB RAM, under Windows 98 SE operating system.

5.1 Results on the Calgary Corpus

An experimental comparison of our PPMVC variants with PPMd, Shkarin's implementation of the PPMII, was performed on the Calgary Corpus files. The results presented in Table 1 indicate that, as we expected, PPMVC3 gives the best compression, PPMVC1 is the fastest but also has worst compression of the three variants, and PPMVC2 is in between. From the point of practical use (also taking into account the memory requirements), PPMVC2 seems to be the best choice.

Now we try to compare PPMII and PPMVC2. In order 4 we have a significant compression improvement, about 0.058 bpc on average. The improvement can be observed for each file except `obj1`. Not surprisingly, the improvement in order 8 (0.011 bpc) is lower than in order 4. For `obj1` some loss is seen again. Significant gain is observed for the last three files (`progl`, `progp`, `trans`). It suggests that only these files contain many long repeated strings. There is insignificantly little average gain in order 16. High-order PPMII predicts so well that there is hardly any space for improvement.

Two main conclusions can be drawn from those results. First, the Calgary Corpus files are too small to get better compression performance from the introduced PPMII modification. Second conclusion is that PPMVC2 in order 8 (2.103) can almost match PPMII's result in order 16 (2.099). Order-8 PPMVC2 needs less memory than PPMII in order 16. Also PPMVC2 results in order 4 (2.153) are ones of the best reported.

5.2 Results on the Silesia Corpus

Silesia Corpus [6, 7] is a recently developed set of large files (each greater than 5 MB) that seems to avoid several shortcomings of the Calgary Corpus (e.g., the excess of English textual data). The tests on Silesia Corpus files were carried out with the same specified memory limit (200 MB). For the comparison we took many algorithms described in the literature, implementations of which are publicly available. Due to scarcity of space, we only list their names without presenting much additional information.

	order 4				order 8				order 16			
	PPMd	VC1	VC2	VC3	PPMd	VC1	VC2	VC3	PPMd	VC1	VC2	VC3
bib	1.832	1.806	1.783	1.778	1.727	1.730	1.726	1.726	1.725	1.728	1.725	1.726
book1	2.246	2.247	2.245	2.245	2.185	2.185	2.185	2.185	2.188	2.188	2.188	2.188
book2	1.957	1.943	1.933	1.931	1.833	1.833	1.832	1.832	1.830	1.831	1.830	1.830
geo	4.352	4.350	4.349	4.349	4.339	4.340	4.339	4.339	4.335	4.336	4.335	4.335
news	2.324	2.281	2.262	2.256	2.201	2.195	2.189	2.188	2.188	2.187	2.186	2.186
obj1	3.539	3.565	3.545	3.547	3.534	3.541	3.537	3.537	3.532	3.536	3.534	3.534
obj2	2.363	2.257	2.250	2.243	2.195	2.178	2.177	2.174	2.161	2.161	2.160	2.160
paper1	2.257	2.237	2.233	2.230	2.195	2.194	2.194	2.193	2.190	2.191	2.191	2.191
paper2	2.227	2.223	2.217	2.217	2.178	2.177	2.178	2.177	2.175	2.176	2.176	2.176
pic	0.782	0.768	0.759	0.760	0.756	0.747	0.743	0.743	0.756	0.745	0.744	0.745
progc	2.291	2.263	2.257	2.258	2.212	2.213	2.210	2.209	2.201	2.203	2.202	2.202
progl	1.657	1.546	1.523	1.521	1.467	1.453	1.446	1.444	1.438	1.439	1.435	1.434
progp	1.659	1.519	1.499	1.491	1.521	1.460	1.458	1.451	1.451	1.439	1.439	1.438
trans	1.460	1.299	1.285	1.274	1.255	1.234	1.230	1.225	1.218	1.220	1.217	1.216
avg.	2.210	2.165	2.153	2.150	2.114	2.106	2.103	2.102	2.099	2.099	2.097	2.097
total ctime [s]	2.03	2.63	2.85	3.90	3.13	3.74	3.85	4.18	3.57	4.06	4.12	4.12

Table 1: Comparison of PPMd and the three variants of PPMVC on the Calgary Corpus files. Compression ratios in bits per characters. The last row contains total compression times for all the files.

The BWT-based compressors used in the comparison are:

- ABC 2.4;
- bzip2 1.0.2 [14], blocks of 900 kB;
- UHBC 1.0 [10], options: -m2, blocks of 5 MB.

Selected PPM implementations:

- PPMZ2 0.8 [3];
- UHARC 0.4 [9];

- PPMd var. I [16];
- PPMonstr var. I [16].

Non-standard approach:

- ACB 2.00c [4];
- PAQ1 [11].

UHARC 0.4 was tested with two sets of switches. In one case, a pure literal PPM model was used. In the other case, the PPM model was strengthened with a string matching model (both worked in parallel). UHARC’s string matching algorithm may be classified to Bloom’s LZP family [1].

	PPMd o4	PPMd o8	PPMd o16	PPMVC2 o4	PPMVC2 o8	PPMVC2 o16	PPMonstr o4	PPMonstr o8	PPMonstr o16
dickens	1.963	1.780	1.803	1.940	1.774	1.825	1.928	1.715	1.698
mozilla	2.652	2.493	2.471	2.575	2.484	2.480	2.237	2.095	2.078
mr	1.860	1.851	1.856	1.860	1.851	1.856	1.835	1.823	1.831
nci	0.668	0.523	0.438	0.550	0.446	0.375	0.574	0.452	0.398
ooffice	3.403	3.268	3.257	3.354	3.263	3.256	2.942	2.839	2.825
osdb	1.912	1.883	1.880	1.947	1.903	1.886	1.848	1.820	1.819
reymont	1.526	1.292	1.224	1.529	1.295	1.226	1.413	1.180	1.107
samba	1.708	1.475	1.388	1.524	1.410	1.388	1.504	1.314	1.238
sao	5.236	5.261	5.270	5.236	5.261	5.270	4.763	4.768	4.773
webster	1.482	1.233	1.300	1.465	1.234	1.317	1.402	1.158	1.151
x-ray	3.633	3.633	3.633	3.633	3.633	3.633	3.578	3.583	3.583
xml	0.908	0.653	0.567	0.700	0.592	0.542	0.803	0.604	0.534
avg.	2.246	2.112	2.091	2.193	2.096	2.088	2.069	1.946	1.920
total ctime [s]	224.5	289.1	329.0	299.2	360.7	399.2	1086.0	1286.0	1553.5

	ABC	ACB	bzip2	gzip	LZMA	PAQ1	PPMZ2	UHARC 0.4	UHA4- LZP	UHBC
dickens	1.840	1.931	2.197	3.023	2.222	1.767	1.861	1.983	1.946	1.793
mozilla	2.608	2.436	2.798	2.967	2.112	2.391	–	2.404	2.250	2.513
mr	1.795	2.007	1.959	2.948	2.206	1.937	1.909	2.153	2.131	1.735
nci	0.346	0.368	0.432	0.712	0.422	0.517	0.394	0.680	0.426	0.303
ooffice	3.356	3.264	3.722	4.019	3.159	3.193	3.457	3.200	3.091	3.330
osdb	1.908	2.075	2.223	2.948	2.260	2.190	1.941	2.028	2.116	1.789
reymont	1.244	1.313	1.504	2.198	1.590	1.298	1.300	1.532	1.521	1.201
samba	1.562	1.453	1.685	2.002	1.402	1.434	1.495	1.657	1.387	1.479
sao	5.197	5.199	5.450	5.877	4.888	4.860	5.170	4.997	4.998	5.176
webster	1.395	1.441	1.668	2.327	1.664	1.251	–	1.502	1.463	1.298
x-ray	3.618	4.043	3.824	5.700	4.242	3.842	3.809	4.181	4.181	3.461
xml	0.604	0.577	0.660	0.991	0.680	0.611	0.545	0.911	0.618	0.579
avg.	2.123	2.176	2.343	2.976	2.237	2.108	–	2.269	2.177	2.054
total ctime [s]	363.6	4881.4	274.6	118.4	1181.4	3299.0	12149.6	685.3	621.7	571.9

Table 2: Comparison of selected compressors on the Silesia Corpus files. Compression ratios in bits per characters. The last row contains total compression times for all the files.

For the comparison we chose compressors, which, to the best of our knowledge, do not make any data specific filtering or other “tricks”. Some of the tested compressors (e.g., UHARC) offered a possibility for turning off the “tricks”, which was chosen for the experiments.

PPMZ2 couldn't have finished compressing the two largest files from the Silesia Corpus (it stopped responding).

As explained earlier, in this test we have experimented only with PPMVC2. The compression speed is slightly worse than that of PPMd (most of the loss is seemingly caused by use of another compiler). PPMVC2's average gain over PPMd is 0.053 bpc in order 4, 0.019 bpc in order 8 and 0.003 bpc in order 16. Almost all this gain, especially in higher orders, is achieved thanks to the two files: `nci` and `xml`. These files are very redundant and PPMVC eliminates the previously mentioned PPM deficiency. Surprisingly, the results for `nci` and `xml` are in most cases even better than PPMonstr's respective ones (implementation of “complicated PPMII” algorithm [16], a more complex and much slower version of PPMII).

6 Conclusions

We presented a string matching extension of the PPM algorithm, with results of an implementation based on Shkarin's PPMII. The additional memory requirements for the necessary data structures are moderate. The main drawback of the existing implementation is prolonged compression/decompression time. The string matching is performed on variable-length order context basis. The described algorithm significantly improves the compression performance of the “pure” character oriented PPM, especially in lower orders (up to 8). The idea seems to be a practical “complementary tool” for traditional PPM models, especially useful for redundant data.

Acknowledgements

We thank Dmitry Shkarin for making the sources of his excellent PPMd public and for useful hints. Also thanks to Uwe Herklotz for many useful suggestions and providing details on his UHARC algorithm, to Sebastian Deorowicz for discussing the implementation issues, and to Maxim Smirnov for valuable help in improving this manuscript.

References

- [1] C. Bloom, “LZP: A new data compression algorithm,” in *Proc. Data Compression Conference*, p. 425, 1996.
- [2] C. Bloom, “Solving the problems of context modeling,” <http://www.cbloom.com/papers/ppmz.zip>, 1998.
- [3] C. Bloom, PPMZ2—High Compression Markov Predictive Coder, <http://www.cbloom.com/src/>, 1999.
- [4] G. Buyanovsky, ACB 2.00c, ftp://ftp.elf.stuba.sk/pub/pc/pack/acb_200c.zip, April 1997.
- [5] J. G. Cleary, W. J. Teahan, and I. H. Witten, “Unbounded length contexts for PPM,” in *Proc. Data Compression Conference*, pp. 52–61, 1995.
- [6] S. Deorowicz, “Universal lossless data compression algorithms,” Ph.D. thesis, Silesian University of Technology, Gliwice, Poland, June 2003.

- [7] S. Deorowicz, Silesia compression corpus, <http://www-zo.iinf.polsl.gliwice.pl/~sdeor/corpus.htm>.
- [8] J.-L. Gailly, gzip 1.2.4, <http://www.gzip.org/>, August 1993.
- [9] U. Herklotz, UHARC 0.4, <ftp://ftp.elf.stuba.sk/pub/pc/pack/uharc04.zip>, December 2001.
- [10] U. Herklotz, UHBC 1.0, <ftp://ftp.elf.stuba.sk/pub/pc/pack/uhbc10.zip>, June 2003.
- [11] M. V. Mahoney, “The PAQ1 data compression program,” <http://www.cs.fit.edu/~mmahoney/compression/paq1.pdf>, 2002.
- [12] I. Pavlov, 7-Zip 2.30 Beta 32, <http://www.7-zip.org>, May 2003.
- [13] E. Roshal, RAR 3.00, <http://www.rarsoft.com>, May 2002.
- [14] J. Seward, bzip2 1.0.2, <http://sources.redhat.com/bzip2/>, January 2002.
- [15] D. Shkarin, “PPM: one step to practicality,” in Proc. *Data Compression Conference*, pp. 202–211, 2002.
- [16] D. Shkarin, PPMd and PPMonstr, var. I, <http://compression.graphicon.ru/ds/>, April 2002.
- [17] P. Skibiński, PPMVC 1.0 sources, <http://www.ii.uni.wroc.pl/~inikep/research/PPMVC10.zip>, July 2003.