SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. (2011)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.1070

SHORT COMMUNICATION

PPM compression without escapes

P. M. Fenwick*, †

Department of Computer Science, The University of Auckland, Auckland, New Zealand

SUMMARY

A significant cost in PPM data compression (and often the major cost) is the provision and efficient coding of escapes while building contexts. This paper presents some recent, preliminary, work on a new technique for eliminating escapes in PPM compression, using bit-wise compression with binary contexts. It shows that PPM without escapes can achieve averages of 2.5 bits per character on the Calgary Corpus and 2.2 bpc on the Canterbury Corpus, both values comparing well with accepted good compressors. Copyright © 2011 John Wiley & Sons, Ltd.

Received 24 September 2010; Revised 14 January 2011; Accepted 24 January 2011

KEY WORDS: PPM; no escapes; bit-wise compression

1. INTRODUCTION

Computer algorithms may be characterized by the algorithm (simple or complex), the data structures (simple or complex) and again data structures (large or small). The work described in this paper grew out of considering how data compression might be achieved by combining a simple algorithm and a simple but possibly large data structure.

A long-time standard in text compression is the PPM family ('Prediction by Partial Matching'), initially proposed by Cleary and Witten [1]. Significant improvements were given by Bloom [2], Cleary *et al.* [3] and later in the PAQ family of compressors [4, 5]. Another important prior-art compressor using bit encoding is Context Tree Weighting (CTW [6, 7]). What is shared by all these 'improvements' is the increasing complexity of the algorithms, with ever-more complex models for statistical prediction and even machine-learning techniques such as neural nets.

This paper in contrast examines some compressors with simple, even trivial, algorithms and no statistical or other optimization. The paper originated from noting that a major part of the cost in PPM compression seemed to be associated with the handling of escapes. It follows that a PPM-style compressor with *no* escapes might achieve a useful improvement in compression compared with conventional methods. While not actually giving an improvement, the current work introduces techniques that give reasonable compression with no explicit escapes. It could well form the basis for further study.

*Correspondence to: P. M. Fenwick, Department of Computer Science, The University of Auckland, Auckland, New Zealand.
†E-mail: p.fenwick@auckland.ac.nz

2. PPM COMPRESSION

To summarize its operation, a PPM compressor builds a table of 'contexts'. Each context is defined by *n* consecutive symbols from the data stream (giving an 'order-*n*' context) and is accompanied by symbols that have been seen to follow occurrences of the context. As a new symbol may occur at any time, some means must exist to allow augmentation of the context symbols, and signalling the arrival to the decoder. Usually this means that every context must include an *escape* symbol whose use signals that the coder/decoder must move to a lower order context, which is usually more populous. Many cases require a sequence of escapes to drop down even to order 1 or order 0.

The probability of the escape relative to the other symbols is crucial, but can only be estimated; it is known in the literature as the *zero frequency problem*—what probability should be assigned to a symbol which has not yet been seen? If the escape is given a low probability it will be expensive to emit; a high probability penalizes the emission of the existing, known, symbols.

Some investigations by the author [8], combining Burrows–Wheeler and PPM compression, showed that a large part of the output stream was concerned with establishing contexts (in effect handling escapes) rather than actually coding symbols—about 50% for order 4 contexts and perhaps 90% for order 12. The author has therefore looked at the possibility of reducing the number of escapes or even omitting them entirely.

This current paper presents one aspect of this investigation. In contrast to most other developments, it emphasizes very simple modelling and algorithms, even at the cost of relatively high memory usage.

3. COMPRESSORS AND RESULTS

This section will present a sequence of three compressors to illustrate the development of the final technique. The results are summarized, for the Calgary and Canterbury Corpora, in Table I, but first including three other compressors for comparison and later one other test file.

3.1. GZIP

Standard GZIP with default parameters, given here as a widely available production compressor.

3.2. CTW

The Context-Tree Weighting algorithm (v0.1) included as an example of a high-performance bit encoding algorithm, run with default parameters.

3.3. PPMref

This is a simple PPM compressor, written by the author as a testbed. It uses full exclusions as a simple form of blending, but has no optimizations. The zero-frequency problem is basically ignored; an escape is just another symbol, always included in each context, its reference count combining with those of other symbols to control data coding. Its one unusual feature is that it is an order-5 compressor, with order-4 omitted; order-5 escapes go directly to order-3. (See the discussion toward the end of Section 3.6.) Its compression is very similar to that of PPM*[3], and within about 7% of the far more complex Context-Tree Weighting [6, 7]. Although non-standard, it is included here as an example of what is possible with simple unoptimized PPM.

3.4. BWC (BitWise compressor)

The simplest arithmetic coding model is for a binary alphabet, with two counts. If the counts will each fit into a single byte, we need only two bytes per model and a data structure of millions of models is quite feasible.

The BWC uses a set of 1 million (2^{20}) binary models each with two 1-byte counters and held as a simple linear array occupying in all 2 MB. This array is indexed by the previous 20 bits of the data stream, with no regard for byte boundaries, giving a binary PPM compressor of order 20.

The result is a surprisingly good compressor, generally within 10–15% of GZIP. There is little benefit in increasing the order; in fact `geo` is best with lower orders. Increasing the counts to 2 bytes gives only modest improvement (say 1%), while doubling the storage cost.

3.5. BWCV (BitWise compressor variable order)

A PPM encoder issues an escape when it must process a symbol which is not present within the context for the current order. This requirement does not apply to a binary order, which *always* contains both valid symbols. But we can also escape if the context itself is absent or invalid. With a binary context this is easily achieved by initializing both counts to zero. An invalid (zero) context can then force an automatic escape to a lower order, *with no explicit escape code emission*.

This is done in the BWCV compressor. All models are initialized to $\{0, 0\}$, except for the zero-order context, set to $\{1, 1\}$. The coder tracks down from the highest order (still 20 and binary) until a valid (non-zero) context is found, from which the bit can be emitted. The coder then traces back through the bypassed orders, validating each by first setting its counts to $\{1, 1\}$ and then incrementing the appropriate count (by 4). The decoder of course mirrors these actions, following the coder, governed by just by the validity of the contexts and with no explicit signalling. The result is a small improvement over the BWC compressor for most files.

The result for `pic` is especially good, but perhaps not surprisingly as this is essentially the bit-stream data which is forced into a byte representation.

3.6. PPMbit (PPM BitWise)

The two previous compressors (with binary order=20) are equivalent to byte-wise PPM of order only 2.5. While this can be extended by increasing the binary order, even as far as 28 with current RAM sizes, some other aspects become important

1. The models array becomes increasingly sparse and inefficient in space utilization.
2. The array becomes quite time-consuming to initialize, especially for sizes of 100 MB (say binary orders of 26–27). The initialization can easily overwhelm the compression proper.
3. Performance suffers if the models do not fit into a cache or, for *very* large models, into real memory.
4. There is anyway little gain with these relatively modest increases in coding order.

These problems all follow from the essentially simple, even crude, implementation with all contexts held in a simple linear array. Typically only 15–20% of the 2^{20} entries are actually used. We could increase the storage efficiency by storing the models in a hash table or a similar structure. Instead we combine the byte-oriented context management of the PPMref compressor with the bit-wise coding within each byte from Section 3.5, with results shown in the PPMbit column of Table I. The byte contexts are determined as usual for PPM. Each context contains an array of 256 binary models, all except for the zero-order byte context initialized to 0 (invalid). Each bit is processed as with the BWCV compressor, dropping down the *byte* order as needed to find a valid context for that bit. It is quite usual to find one byte being handled by bit models at several different byte-order contexts, just as consecutive bytes may be handled by different orders with normal PPM.

In essence the array of models for each byte is indexed by initially 0, then the left-most bit, then the left-most two bits and so on until the right-most bit uses a model selected by the preceding 7 bits. This gives a small bit-wise PPM structure within each byte. A naive implementation might just use initial bits as indices, but this makes all leading zeros map to the same model. We, therefore, add a prefix 1 bit to the byte code to separate out the leading zeros. For example, processing the letter 'J' (01001010) uses the sequence of indices into the array of models (decimal values in parentheses)—1(1), 10(2), 101(5), 1010(10), 10100(20), 101001(41), 1010010(82) and 10100101(165).

With this scheme the first binary model (1, to process bit 7) is used for every bit, whereas the usage frequency decreases for later bits as the coding tree branches out. The leftmost bit (bit 7) is, therefore, given a 'small' increment of 5 and the rightmost (bit 0) a larger increment of 7, with linear interpolation for intermediate bits.

We find that often 60–80% of the highest order contexts are deterministic, with only a single byte. As, storage-wise, it is most inefficient to handle these with the full set of 256 models, we use a different technique for deterministic contexts.

1. New contexts are handled by normal bit encoding, but all bits are coded at some lower level because the highest order contexts do not yet exist. (The new contexts are created by back-tracking up the orders, as in the variable-order compressor above—Section 3.5.) The bits are assembled into a single symbol held within the context and the context marked as 'deterministic'.
2. Contexts with two or more symbols are processed as normal, processing each bit in sequence.
3. Deterministic contexts, with only a single symbol, are handled specially. In coding, if the current symbol matches that already held, we emit a 'copy' flag and nothing more. The decoder, also knowing that the context contains a unique symbol, reads this flag and copies one symbol to the output.
4. If the symbol to be coded does not match that in the deterministic context, the copy flag signals that full 8-bit coding is needed. The previously deterministic context then becomes (at both coder and decoder) a normal, multiple, context, as in (2) above.

This handling of deterministic contexts gives a useful saving of memory and also a slight (1–2%) improvement in compression. Typically 80–90% of the highest order context symbols are copied rather than coded.

Much as with the PPMref compressor, the best results here use orders 0, 1, 2 and 4, omitting order 3. Most of the 'filtering' or elimination of symbols from the context occurs between orders 1 and 2 (at least for text files). It seems to be more effective to improve the count statistics at the lower order (2), rather than emit bits from the higher order (3) which, being used less frequently, accumulates poorer statistics.

This compressor is nearly as good as PPMref on most files, and compares well with GZIP. The final column in Table I shows the memory requirements for each of the test files, using the final PPMbit compressor.

Finally, most of the PPMbit results are within 10–15% of CTW values (some to 21%, but better for `excl`). The PPMbit compressor, with no optimization at all, thus compares well against other compressors with extensive optimization.

But why is this compressor, with no escapes, generally inferior to the otherwise comparable PPMref, with conventional escapes? Two reasons are suggested.

- In PPMref the entire byte has a definite probability and is coded accordingly. In PPMbit the probability gradually develops as bits are processed; it is only the last, rightmost, bit that is truly emitted at an appropriate probability. Earlier bits are effectively blurred among several byte values. In text files about 10% of the emitted bits are coded from data bit 0, increasing to about 20% for bit 4, demonstrating the variation in efficiency. (The three high-order bits identify letters, digits, etc. and behave quite differently.) A similar, but smaller, trend appears in the binary files.
- And also, the author's experience suggests that there is always some inefficiency in each stage of arithmetic coding, despite its asymptotic ideal. If this is so, coding a byte as 8 individual bits rather than a single entity will again degrade the compression.

Table I includes comparative results for the (small!) 100 MB file often used with the PAQ compressor family. In 'their' terms, the output file from PPMbit is about 26.2 MB, compared with 17–20 MB for the best of the far more complex PAQ compressors. But the compression compares very well with the GZIP compressor and even CTW. This section includes compression times for `enwik8` on a 2.2 GHz X86 processor.

Table I. Calgary and Canterbury Corpus Results.

	GZIP	CTW	PPMref ord 1235	BWC order 20	BWCV var order	PPMbit bpc	PPMbit MB
Calgary Corpus							
bib	2.521	1.832	1.964	2.820	2.769	2.078	3.9
book1	3.261	2.180	2.307	2.890	2.873	2.461	14.2
book2	2.707	1.891	2.019	2.849	2.841	2.150	12.6
geo	5.351	4.531	4.796	5.939	5.719	5.039	15.4
news	3.073	2.350	2.440	3.400	3.383	2.625	14.4
obj1	3.840	3.716	3.888	4.594	4.995	4.026	3.7
obj2	2.646	2.398	2.533	3.529	3.618	2.638	14.2
paper1	2.796	2.289	2.403	3.195	3.135	2.603	3
paper2	2.896	2.228	2.351	3.000	2.950	2.573	3.7
pic	0.880	0.796	0.847	0.819	0.808	0.877	6.4
progc	2.681	2.335	2.470	3.248	3.187	2.622	2.6
progl	1.817	1.646	1.798	2.489	2.448	1.883	2.4
progp	1.822	1.677	1.799	2.488	2.435	1.872	2
trans	1.621	1.442	1.602	2.532	2.520	1.633	2.9
Average	2.708	2.237	2.373	3.128	3.120	2.506	
Canterbury Corpus							
bible	2.354	1.506	1.611	2.321	2.325	1.696	12.5
csrc	2.253	1.982	2.162	2.874	2.808	2.195	0.8
ecoli	2.313	1.940	1.948	2.010	2.008	2.005	0.2
excl	1.600	1.008	1.248	1.608	1.604	0.953	8.1
fax	0.880	0.796	0.847	0.819	0.808	0.877	6.4
html	2.600	2.303	2.412	3.238	3.144	2.515	1.7
lisp	2.664	2.357	2.624	3.413	3.291	2.628	0.4
man	3.318	2.939	3.085	4.230	4.073	3.286	0.6
play	3.128	2.322	2.475	2.911	2.858	2.713	5.3
poem	3.241	2.185	2.328	2.789	2.772	2.463	9
sprc	2.703	2.567	2.710	3.545	3.472	2.766	3.5
tech	2.715	1.832	1.968	2.747	2.731	2.099	8.2
text	2.863	2.075	2.210	2.768	2.727	2.394	5
world192	2.344	1.432	1.555	2.803	2.798	1.743	28.4
Average	2.498	1.946	2.085	2.720	2.673	2.167	
enwik8	2.801	2.036	1.934	2.965	2.978	2.099	78.2
PC time (s)	12	1770	220	49	61	100	

4. CONCLUSIONS

This paper has shown that an escape-free PPM compressor using simple algorithms and bit-wise coding rather than byte-wise coding can achieve compression well within the range of accepted ‘good’ compressors, even with no statistical analysis or other optimization of the text.

It is presented as a preliminary work, illustrating the feasibility of escape-free PPM. But perhaps it also demonstrates the diminishing returns of improving the escape mechanism; indeed escape handling might not be a limiting factor.

In retrospect, there seems to be a ‘swings and roundabout’ situation with respect to escapes; if escapes are eliminated, their cost is very often transferred to some other part of the coding. For example, Burrows–Wheeler compression eliminates escapes, but only at the cost of also eliminating most knowledge of the contexts. The contexts can be recovered to some extent, but only with considerable difficulty. (Refer [8] for discussion.) In the present situation, coding uses the preceding bytes as a context, but largely ignores the immediate context of the current byte, giving a less-efficient coding of the earlier bits of each byte.

Again, the present compressor follows traditional PPM in using only a first-order context of the preceding symbols. In contrast the very best compressors use extensive and complex statistical analyses to obtain more efficient prediction and coding of the bytes (or perhaps bits), an approach deliberately avoided in the current work. It is hoped some later author might consider combining the present techniques of escape elimination with improved estimation of the symbols themselves.

ACKNOWLEDGEMENTS

The author appreciates the very constructive comments from the referees.

REFERENCES

1. Cleary JG, Witten IH. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, *COM-32* 1984; **4**:396–402.
2. Bloom C. Solving the problems of context modeling. *Informally Published Report*, 1988. Available at: http://www.cbloom.com/papers/.
3. Cleary JG, Teahan WJ, Witten IH. Unbounded length contexts for PPM. *Data Compression Conference*, Salt Lake City, UT, 1995; 52–61.
4. Mahoney M. Available at: http://mattmahoney.net/dc/cs200516.pdf [2010].
5. Mahoney M. Available at: http://mattmahoney.net/dc/cs200516.pdf (see comment in PAQ8 source) [2010].
6. Willems FMJ, Shtarkov YM, Tjalkens TJ. The context-tree weighting method: Basic properties. *IEEE Transactions on Information Theory* 1995; **IT-41**:653–664.
7. CTW (Context Tree Weighting) website. Available at: http://www.ele.tue.nl/ctw/ [2010].
8. Fenwick P. Burrows-wheeler compression: Principles and reflections. *Theoretical Computer Science* 2007; **387**(3):200–219.