Design and Analysis of Fast Text Compression Based on Quasi-Arithmetic Coding

Paul G. Howard and *Jeffrey Scott Vitter*

To appear in *Information Processing and Management.*

A shorter version appears in the proceedings of the
IEEE Computer Society/NASA/CESDIS Data Compression Conference,
Snowbird, Utah, March 30–April 1, 1993, pages 98–107.

Design and Analysis of Fast Text Compression Based on Quasi-Arithmetic Coding[1]

Paul G. Howard[2]
Department of Computer Science
Brown University
Providence, R.I. 02912–1910

Jeffrey Scott Vitter[3]
Department of Computer Science
Duke University
Durham, N.C. 27706–0129

Abstract

We give a detailed algorithm for fast text compression. Our algorithm, related to the PPM method, simplifies the modeling phase by eliminating the *escape* mechanism and speeds up coding by using a combination of *quasi-arithmetic coding* and Rice coding. We provide details of the use of quasi-arithmetic code tables, and analyze their compression performance. Our *Fast PPM* method is shown experimentally to be almost twice as fast as the PPMC method, while giving comparable compression.

1 Introduction

For compression of text files, the best compression results from the use of high-order models in conjunction with statistical coding techniques. The best compression reported in the literature comes from the PPM (prediction by partial matching) method of Cleary and Witten [3]; the most widely used implementation is Moffat's PPMC. The PPM methods use adaptive context models with a fixed maximum order, and arithmetic coding for the coder.

In this paper we show that we can obtain significantly faster compression with only a small loss of compression efficiency by modifying both the modeling and coding aspects of PPM. The important idea is to concentrate computer resources where they are needed for good compression while using simplifying approximations where they cause only slight degradation of compression performance.

On the modeling side, we eliminate the explicit use of *escape* symbols, we use approximate probability estimation, and we simplify the repeated-symbol-exclusion mechanism. For the coder, we replace the time-consuming arithmetic coding step with various combinations of *quasi-arithmetic coding* and simple prefix codes from the Rice family. Quasi-arithmetic coding, introduced and explained in [6], is a variation of arithmetic coding [11] that uses lookup tables after performing all the arithmetic ahead of time. The computations are done to low precision to keep the table sizes manageable.

In Section 2 we briefly describe the PPM method and our speed-oriented enhancements. In Section 3 we describe our implementation, including a detailed example showing both encoding and decoding using quasi-arithmetic coding. In Section 4 we analyze quasi-arithmetic coding, showing that using it instead of full-precision arithmetic coding causes only a small loss of compression efficiency. In Section 5 we show experimentally that our methods run nearly twice as fast as PPMC, with comparable compression.

2 Prediction by Partial Matching

The Cleary-Witten PPM method. The PPM idea is to maintain contexts of different lengths up to a fixed maximum order o. To encode a new symbol, we check whether the current order-o context has occurred, and if so, whether the new symbol has occurred in that context. If it has, we use arithmetic coding to encode the symbol based on the

[1] A shorter version of this paper appears in the proceedings of the IEEE Computer Society/NASA/CESDIS Data Compression Conference, Snowbird, Utah, March 30–April 1, 1993, 98–107.

[2] Support was provided in part by NASA Graduate Student Researchers Program grant NGT–50420, by a Universities Space Research Association/CESDIS associate membership, and by National Science Foundation grant IRI–9116451.

[3] Work was performed while the author was at Brown University. Support was provided in part by a National Science Foundation Presidential Young Investigator Award with matching funds from IBM and by Air Force Office of Scientific Research grant number F49620–92–J–0515. Additional support was provided by a Universities Space Research Association/CESDIS associate membership.

Order	Context	Symbol	Count	Action
3	nin	–	–	automatic escape
2	in	▪	1	NOT FOUND
		n	1	NOT FOUND, escape
1	n	▪	1	exclude
		n	1	exclude
		i	1	NOT FOUND, escape
0	–	n	4	exclude
		i	3	exclude
		▪	2	exclude
		e	2	NOT FOUND
		t	1	NOT FOUND
		h	1	NOT FOUND
		b	1	NOT FOUND
		g	1	FOUND
−1	Full alphabet not needed.			

(a)

	Symbol
0	▪
1	n
2	i
3	e
4	t
5	h
6	b
7	g
8	*new-symbol*
9	*end-of-file*

(b)

Table 1: Example of PPM operation, maximum coding order $o = 3$. (a) Standard PPM. Suppose we are encoding the short message "in▪the▪beginning" (▪ representing the space character), and that we have coded all but the final 'g'. The current order 3 context, 'nin', has never occurred, so we try order 2. Neither of the symbols that have occurred in the current order-2 context are the one we want, so we explicitly escape to order 1. At order 1 we can exclude '▪' and 'n' since we already checked them at order 2; 'i' is not the letter we want, so we escape to order 0, the empty context. At order 0 we exclude 'n', 'i', and '▪', and check the others until we come to 'g'. This is the letter we want, so we code it and stop. If the symbol had not yet occurred in the message, we would have escaped to order "−1" which includes the entire alphabet. In this example contexts of all orders have been created or updated after coding each symbol. (b) Concatenated list in Fast PPM at the same point in the coding. It results from combining the lists of various orders and eliminating duplicate symbols. The *new-symbol* and *end-of-file* pseudo-symbols have been added to the end of the list. We code 'g' by indicating 7 NOT-FOUNDs and one FOUND.

current symbol counts in the context. Otherwise, we encode a special *escape* symbol (whose probability must be estimated) and repeat the process with progressively shorter contexts until we succeed in encoding the symbol. (In the shorter contexts we may exclude from consideration symbols that have already been rejected in longer contexts.) If a symbol has never occurred in any context, we escape to a special context containing the entire alphabet (including a special end-of-file symbol, but possibly excluding symbols already rejected), thus ensuring that every symbol can be encoded. Table 1(a) illustrates the coding of one symbol using the PPM method.

The symbols are coded using a multi-symbol arithmetic coder. The probabilities passed to the coder are based on symbol frequency counts, periodically scaled down to exploit locality of reference. At least seven different methods have been used to estimate the *escape* probability [1,3,6,8,10]; Moffat's PPMC [8] is the most widely used, although our PPMD method [6] consistently gives about one percent better compression on text files.

Fast PPM. We observe that the use of arithmetic coding guarantees good compression but runs slowly: the multi-symbol version used in PPMC requires two multiplications and two divisions for each symbol coded, including *escapes*. We also note that often the PPM method predicts very well. When we compress text files using a maximum order of 3 or more, we find that the symbol that actually occurs is the most frequent symbol in the longest available context more than half the time, as seen in Table 2. This implies that the *escape* mechanism is not needed very often. (This is one reason for the observations by Cleary, Witten, and Bell that the choice of *escape* probability makes little difference in the amount

File	Maximum order				
	1	2	3	4	5
bib	30.3	47.0	58.7	62.6	63.5
book1	26.9	39.4	48.8	53.2	54.2
book2	24.5	39.7	52.9	59.4	61.2
news	22.6	37.9	50.4	55.4	56.4
paper1	25.2	42.1	52.6	55.6	56.0
paper2	26.5	41.3	51.4	54.8	55.3
progc	27.9	45.9	54.9	57.0	57.4
progl	31.0	49.3	60.4	64.0	65.3
progp	38.0	56.4	65.8	67.7	68.2
trans	33.6	52.9	65.9	69.7	70.7

Table 2: Probability of finding next symbol in one trial. We show the percentage of symbols that are found as the most probable symbol in the first usable context. The rows represent the ten text files of the Calgary corpus. The columns represent different maximum model orders. The compression program is a version of Fast PPM in which the symbol lists within each context are maintained in approximate frequency count order: when a symbol occurs, its count is compared with that of its predecessor in the list; if the current symbol's count is greater than or equal to that of its predecessor, the two symbols are transposed in the list. For models of maximum order 3, 4, or 5, we find the current symbol in the first position of the longest context more than half the time.

of compression obtained.) Finally, we recall that arithmetic coding significantly outperforms prefix codes like Huffman coding only when the symbol probabilities are highly skewed.

In the methods presented here, we eliminate the *escape* mechanism altogether. First we concatenate the symbol lists of the current contexts of various orders, beginning with the longest, as shown in Table 1(b). (Of course the concatenation is only conceptual. In practice we simply search through the context's lists, moving to the next list when one is exhausted and stopping when we find the current symbol.) To avoid wasting code space, we exclude all but the first occurrence of repeated symbols using the fast exclusion mechanism described in Section 3.

We must identify the current symbol's position within the concatenated list. We choose one of a number of related methods, our choice depending on the speed and compression required. The idea is to use binary quasi-arithmetic coding to encode NOT-FOUND/FOUND decisions for the symbols with highest probability, then if necessary to use a simple prefix code (in particular, a Rice code) to encode the symbol's position in the remainder of the list. For maximum speed, we can eliminate the quasi-arithmetic coding step altogether, while for maximum compression we can eliminate the prefix code, using only a series of binary decisions to identify each symbol. Using quasi-arithmetic coding for just the first symbol in the longest context is a good practical choice, as is using quasi-arithmetic coding until the FOUND probability falls below a specified threshold. Lelewer and Hirschberg [5] also use the idea of coding a symbol's position within a PPM context list.

Quasi-arithmetic coding. In arithmetic coding, we subdivide the real interval $[0, 1)$, the lengths of the subdivisions being proportional to the probabilities of the events that can occur, then select the subinterval corresponding to the event that actually occurs. We recursively repeat the subdivision and selection process for all input symbols. At the end of coding we output enough bits to distinguish the final interval from all other possible final intervals. In practice we use integer arithmetic and subintervals of an integer interval $[0, N)$. We output bits as soon as we know them and expand the interval, allowing us to limit the coding delay and to use finite precision arithmetic. Witten, Neal, and Cleary [11] present a very clear implementation of arithmetic coding; they use a large N for the interval, namely $N = 65{,}536$. In [6] we introduce *quasi-arithmetic coding*, a reduced-precision version of the Witten-Neal-Cleary implementation of arithmetic coding. Our idea is to do all the arithmetic ahead of time and to store the results in lookup tables. Since the number of coder states is $3N^2/16$, if we choose a small enough value for N, the number of coder states will be small enough to permit keeping all the lookup tables in memory. Table 3 is the entire coding table for $N = 8$; in practice somewhat larger values of N give slightly better results.

Start state	Probability of *0* input	*0* input Out	Next state	*1* input Out	Next state
[0, 8)	0.000 – 0.182	**000**	[0, 8)	–	[1, 8)
	0.182 – 0.310	**00**	[0, 8)	–	[2, 8)
	0.310 – 0.437	**0**	[0, 6)	–	[3, 8)
	0.437 – 0.563	**0**	[0, 8)	**1**	[0, 8)
	0.563 – 0.690	–	[0, 5)	**1**	[2, 8)
	0.690 – 0.818	–	[0, 6)	**11**	[0, 8)
	0.818 – 1.000	–	[0, 7)	**111**	[0, 8)
[0, 7)	0.000 – 0.208	**000**	[0, 8)	–	[1, 7)
	0.208 – 0.355	**00**	[0, 8)	–	[2, 7)
	0.355 – 0.500	**0**	[0, 6)	–	[3, 7)
	0.500 – 0.645	**0**	[0, 8)	**1**	[0, 6)
	0.645 – 0.792	–	[0, 5)	**1f**	[0, 8)
	0.792 – 1.000	–	[0, 6)	**110**	[0, 8)
[0, 6)	0.000 – 0.244	**000**	[0, 8)	–	[1, 6)
	0.244 – 0.415	**00**	[0, 8)	**f**	[0, 8)
	0.415 – 0.585	**0**	[0, 6)	**f**	[2, 8)
	0.585 – 0.756	**0**	[0, 8)	**10**	[0, 8)
	0.756 – 1.000	–	[0, 5)	**101**	[0, 8)
[0, 5)	0.000 – 0.293	**000**	[0, 8)	–	[1, 5)
	0.293 – 0.500	**00**	[0, 8)	**f**	[0, 6)
	0.500 – 0.707	**0**	[0, 6)	**ff**	[0, 8)
	0.707 – 1.000	**0**	[0, 8)	**100**	[0, 8)
[1, 8)	0.000 – 0.208	**001**	[0, 8)	–	[2, 8)
	0.208 – 0.355	**0f**	[0, 8)	–	[3, 8)
	0.355 – 0.500	**0**	[2, 8)	**1**	[0, 8)
	0.500 – 0.645	–	[1, 5)	**1**	[2, 8)
	0.645 – 0.792	–	[1, 6)	**11**	[0, 8)
	0.792 – 1.000	–	[1, 7)	**111**	[0, 8)
[1, 7)	0.000 – 0.244	**001**	[0, 8)	–	[2, 7)
	0.244 – 0.415	**0f**	[0, 8)	–	[3, 7)
	0.415 – 0.585	**0**	[2, 8)	**1**	[0, 6)
	0.585 – 0.756	–	[1, 5)	**1f**	[0, 8)
	0.756 – 1.000	–	[1, 6)	**110**	[0, 8)
[1, 6)	0.000 – 0.293	**001**	[0, 8)	**f**	[0, 8)
	0.293 – 0.500	**0f**	[0, 8)	**f**	[2, 8)
	0.500 – 0.707	**0**	[2, 8)	**10**	[0, 8)
	0.707 – 1.000	–	[1, 5)	**101**	[0, 8)
[1, 5)	0.000 – 0.369	**001**	[0, 8)	**f**	[0, 6)
	0.369 – 0.631	**0f**	[0, 8)	**ff**	[0, 8)
	0.631 – 1.000	**0**	[2, 8)	**100**	[0, 8)
[2, 8)	0.000 – 0.244	**010**	[0, 8)	–	[3, 8)
	0.244 – 0.415	**01**	[0, 8)	**1**	[0, 8)
	0.415 – 0.585	**f**	[0, 6)	**1**	[2, 8)
	0.585 – 0.756	**f**	[0, 8)	**11**	[0, 8)
	0.756 – 1.000	–	[2, 7)	**111**	[0, 8)
[2, 7)	0.000 – 0.293	**010**	[0, 8)	–	[3, 7)
	0.293 – 0.500	**01**	[0, 8)	**1**	[0, 6)
	0.500 – 0.707	**f**	[0, 6)	**1f**	[0, 8)
	0.707 – 1.000	**f**	[0, 8)	**110**	[0, 8)
[3, 8)	0.000 – 0.293	**011**	[0, 8)	**1**	[0, 8)
	0.293 – 0.500	**ff**	[0, 8)	**1**	[2, 8)
	0.500 – 0.707	**f**	[2, 8)	**11**	[0, 8)
	0.707 – 1.000	–	[3, 7)	**111**	[0, 8)
[3, 7)	0.000 – 0.369	**011**	[0, 8)	**1**	[0, 6)
	0.369 – 0.631	**ff**	[0, 8)	**1f**	[0, 8)
	0.631 – 1.000	**f**	[2, 8)	**110**	[0, 8)

Table 3: Complete quasi-arithmetic coding code table for $N = 8$, based on the arithmetic coding method described by Witten, Neal, and Cleary. The initial state is $[0, 8)$. An **f** in an "Out" (output) column indicates that the *bits-to-follow* count should be incremented. Within a given state we choose the row based on the probability of a *0* input; the probability ranges are calculated according to Equation (1).

Rice codes. Because a quasi-arithmetic coder must encode a number of binary decisions, a text coder that uses quasi-arithmetic coding alone can take almost as long as PPMC. By encoding a number of decisions at once, however, we can speed up the coder. Rice codes [9] are eminently suitable for encoding a number of NOT-FOUND decisions followed by a single FOUND decision.

Each Rice code has a non-negative integer parameter k. We encode a non-negative integer n by outputting $\lfloor n/2^k \rfloor$ in unary, then outputting $n \bmod 2^k$ in binary. In practice, we divide the binary representation of n into high- and low-order parts, the low-order part consisting of k bits; then we output the high-order part as a unary number, and the low-order part directly as a binary number. For example, to encode $n = 5$ with the Rice code whose parameter $k = 2$, we divide $5_{10} = 101_2$ into $1 \cdot 01$, output **10** (the unary representation of 1, the high order part), and then output **01** (the low order k bits). Several Rice codes are illustrated in Table 4.

Strictly speaking, Rice codes apply to exponential distributions, but in fact they will give good compression for almost any decaying probability distribution. If we keep our symbol lists ordered by frequency count within each context, the concatenated list used to find a symbol will be in decreasing probability order except possibly for bumps where the context lists are joined, so we can use Rice coding to encode symbol positions within the concatenated lists.

n	$k = 0$	$k = 1$	$k = 2$	$k = 3$
0	0·	0·0	0·00	0·000
1	10·	0·1	0·01	0·001
2	110·	10·0	0·10	0·010
3	1110·	10·1	0·11	0·011
4	11110·	110·0	10·00	0·100
5	111110·	110·1	10·01	0·101
6	1111110·	1110·0	10·10	0·110
7	11111110·	1110·1	10·11	0·111
8	111111110·	11110·0	110·00	10·000
9	1111111110·	11110·1	110·01	10·001
⋮	⋮	⋮	⋮	⋮

Table 4: Examples of the beginnings of some Rice codes for several parameter values. In this table a midpoint (·) separates the high-order (unary) part from the low-order (binary) part of each code.

To choose the parameter value k, in each context we maintain a cumulative count for each reasonable parameter value of the number of bits that would have been required if we had always used that parameter value; we then choose the parameter value with the smallest count. This parameter estimation method is presented in detail in [7], where we prove that under reasonable assumptions it produces a code length only $O(\sqrt{t})$ bits in excess of that of the optimal Rice code for a context that occurs t times.

Rice codes are a subset of Golomb codes [4]; in Golomb codes we encode n by outputting $\lfloor n/m \rfloor$ in unary and $n \bmod m$ in binary (adjusted to avoid wasting code space if m is not a power of 2). Since the Rice codes are just the Golomb codes where m is a power of 2, Rice codes are somewhat simpler. Since there are fewer reasonable Rice codes, the parameter estimation technique is faster. We could use Golomb codes in the Fast PPM method; in practice, Rice codes run slightly faster and give about 1 percent worse compression.

3 Implementation

In this section we describe an implementation of the Fast PPM text compression system. We explain the differences in modeling between our method and the PPMC method. Then we discuss the coding phase, particularly quasi-arithmetic coding with precomputed tables. We give an extended example that includes complete coding tables for a small coder.

Data structure for high order models. We use a multiply-linked list structure similar to the vine pointers of Bell *et al.* [2]; the structure is illustrated in Figure 1. In the versions of the Fast PPM system that use Rice coding, we keep the context lists sorted according to frequency count, while in the version that uses only quasi-arithmetic coding we do not reorganize the lists at all.

We delay creating new nodes in order to save time and control the number of nodes present. Every symbol instance appears simultaneously in contexts of all orders from 0 to o, but we do not create nodes for all possible orders. Instead, we create at most one new node for any symbol instance, just one order higher than the one at which the symbol was found. (If it was found at the highest order, we do not create any new nodes.) This procedure runs somewhat counter to a recommendation of Bell *et al.* [2, pages 149–150], but compression does not appear to suffer greatly. We also use a lazy update rule as in [2], updating statistics only for contexts actually searched. In our implementation we allow the model to grow without bound, never deleting nodes or restarting the model. This is a reasonable approach considering the increasing availability of large amounts of inexpensive memory. Hirschberg and Lelewer [5] use a hashing approach to save space in PPM-like models.

Exclusion mechanism. The standard approach for exclusions is to maintain a bit map of alphabet symbols, together with a list of currently excluded symbols to quickly reset the bit map after every symbol. We can make resetting the exclusion map unnecessary by

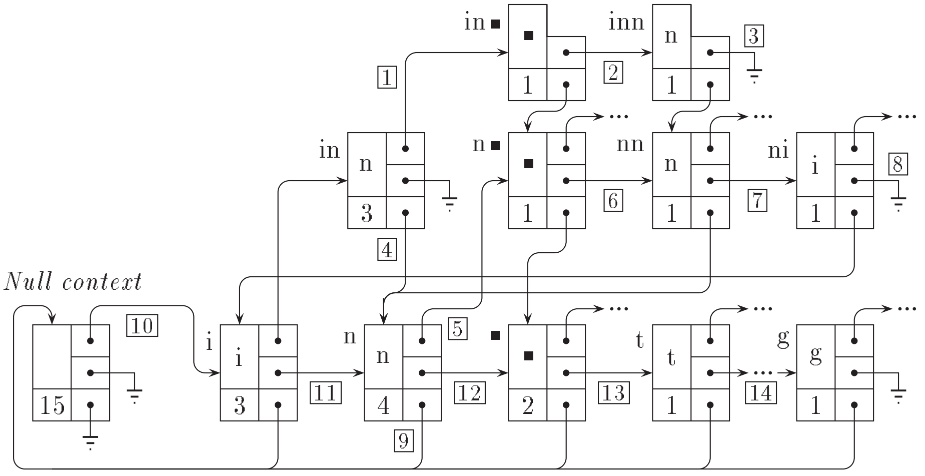

Figure 1: Implementation of part of the multiply-linked list data structure for Fast PPM, maximum order $o = 3$, after coding everything but the final 'g' in "in■the■beginning". Each node except at the highest order is both a node in the list for a certain order (middle link) and the head of a list of the next greater order (upper link). Each node also points to the head of the list of the next smaller order (lower link). For example, the node labeled 'in' is the first (and only) node in the 'i' context; it is the head of the list for the 'in' context, on the top row; and it points to the head of the list for the 'n' context. The numbers in the nodes are frequency counts. To code the last 'g', we would begin at node 'in' and follow the links in the order indicated by the small boxed numbers.

using a time stamp array, with one element for each alphabet symbol. The "time" is the position of the current symbol within the file. When we reject a symbol in the concatenated list, we write the current time in the symbol's position in the time stamp array. If a symbol's entry in the array is the same as the current time, then we must have previously encountered it in the concatenated list for the current symbol, so we can exclude it. We must clear the time stamp array only when the symbol position counter overflows, typically after about $2^{32} \approx 4 \times 10^9$ bytes. When we are using quasi-arithmetic coding for all coding, this mechanism introduces a small inaccuracy in the FOUND/NOT-FOUND probabilities: the NOT-FOUND probabilities will be higher than they should be since they include symbols further down the list that should be excluded. Fortunately the effect is minor.

Coding new symbols and end-of-file. At any point in the coding, the concatenated, duplicate-free context list contains exactly k symbols, where k is the number of distinct alphabet symbols seen so far in the file. To deal with symbols not yet seen in any context, we add a pseudo-symbol whose meaning is "new symbol". When a new symbol occurs, we send the *new-symbol* pseudo-symbol, followed by the uncoded bits of the new symbol. (Using arithmetic coding to identify new symbols requires considerably more work and saves only $k \log_2 n - (\log_2 n! - \log_2 (n-k)!)$ bits for a file with k distinct characters drawn from an n-character alphabet. For $n = 256$ and $k = 100$, this is about 4 bytes.) We also append a second pseudo-symbol to the concatenated list; its meaning is "end-of-file". Hence a sequence of $k + 1$ NOT-FOUNDs (however we choose to code them) means that the file is complete.

Coding. We now explain the coding mechanism and illustrate it with a complete tables and a short example using a small coder. In practice we would use larger tables, but their size remains manageable; the construction and use of the tables follows exactly the same principles. In the example we use $N = 8$, i.e., the full interval is $[0, 8)$. Using $N = 32$ improves compression by about 3.5 percent, and using $N = 128$ gives only another 0.2 percent improvement.

Probability estimation for quasi-arithmetic coding. We use a modification of the scaled-count technique to estimate the FOUND/NOT-FOUND probabilities used by the

Index	Counts		Probability	Transitions	
	F	NF	of F	after F	after NF
$P = 0$	1	4	0.200	$P = 3$	$P = 0$
$P = 1$	1	3	0.250	$P = 4$	$P = 0$
$P = 2$	1	2	0.333	$P = 7$	$P = 1$
$P = 3$	2	4	0.333	$P = 5$	$P = 1$
$P = 4$	2	3	0.400	$P = 8$	$P = 3$
$P = 5$	3	4	0.429	$P = 9$	$P = 4$
$P = 6$	1	1	0.500	$P = 13$	$P = 2$
$P = 7$	2	2	0.500	$P = 11$	$P = 4$
$P = 8$	3	3	0.500	$P = 10$	$P = 5$
$P = 9$	4	4	0.500	$P = 10$	$P = 5$
$P = 10$	4	3	0.571	$P = 11$	$P = 9$
③ $P = 11$	3	2	0.600	$P = 12$	$P = 8$
$P = 12$	4	2	0.667	$P = 14$	$P = 10$
$P = 13$	2	1	0.667	$P = 14$	$P = 7$
① $P = 14$	3	1	0.750	$P = 15$	② $P = 11$
$P = 15$	4	1	0.800	$P = 15$	$P = 12$

Table 5: Probability arrays for quasi-arithmetic coding.

quasi-arithmetic coder. In effect we use small counts for the FOUND and NOT-FOUND events at each decision point; i.e., we keep a count pair F : NF. Only a few bits are used for each count. When either count overflows, we scale both counts downward; the new scaled count pair is the closest to the (unavailable) new count pair, closeness being measured by average excess code length.

In the implementation we denote each possible pair of counts by an index number, and we precompute all the transitions to new count states, including those requiring scaling. In Table 5 we show the correspondence among counts, probabilities, and probability index numbers for a small example coder, as well as all the transitions. For example[4], ① index $P = 14$ corresponds to F : NF $= 3 : 1$ and ② we find that $P = 11$ is the index of the new count state after a NOT-FOUND event, where ③ index $P = 11$ corresponds to F : NF $= 3 : 2$. In the example we allow counts to reach 4; in practice we allow somewhat larger counts (up to 10 or so), and allow some of the unbalanced counts to be larger than the balanced ones. It is quite feasible to store each probability index number in one byte. Only the transition columns are needed by the coder.

Use of quasi-arithmetic coding. We use quasi-arithmetic coding to encode binary decisions, with probabilities (indicated by probability index numbers) supplied by the model. In the implementation we include internal states corresponding to expandable subintervals. The process consists of selecting a new state based on the current event and event probabilities, possibly followed by the output of some bits and a second transition to an unexpandable state. This mechanism makes very efficient use of space in the code tables, allowing us to use a larger full interval and hence to obtain more precise coding and more compression.

We use a pointer into a code table to indicate the state of the coder, corresponding to the current interval in a true reduced-precision arithmetic coder. Table 6 shows a complete code table for $N = 8$ (full interval $[0, 8)$); the initial state is Q_{08}, marked ㉔ in the table. In practice we use a somewhat larger value of N, say 32. We use left subintervals for FOUND decisions and right subintervals for NOT-FOUND decisions.

We illustrate the use of the coder with an example. ④ Suppose we are in state $Q_{17} = [1, 7)$, the F : NF counts are $3 : 1$, indicated by index $P = 14$ ①, and the next decision is NOT-FOUND. ⑤ The W entry for state Q_{17} is W_6 since the width of the interval is 6; ⑥ W_6 is a pointer to one of the five vectors in the delta array (Table 7), the interface between the probability estimator and the coder. (In Section 4 we show how to find the cutoff probabilities between successive values of Δ, which can then be used with Table 5 to compute the delta array.) ⑦ We use $P = 14$ to index into the W_6 vector, and ⑧ find $\Delta = 2$; this is the size of the right subinterval of $[1, 7)$. ⑨ If the decision were FOUND, we would

[4] The small circled numbers key the text to the tables.

Terminal states				Nonterminal states					
	W	H	T		L	R	F	N	Q
㉔ Q_{08}	W_8	H_8	8	Q_{04}	**0**	–	0	1	Q_{08}
Q_{07}	W_7	H_7	7	Q_{03}	**0**	–	0	1	Q_{06}
Q_{06}	W_6	H_6	6	Q_{02}	**0**	**0**	0	2	Q_{08}
Q_{05}	W_5	H_5	5	Q_{01}	**0**	**00**	0	3	Q_{08}
Q_{18}	W_7	H_8	8	Q_{14}	**0**	–	0	1	Q_{28}
④ Q_{17}	W_6 ⑤	H_7 ⑩	7 ㉖	Q_{13}	**0**	–	1	2	Q_{08}
Q_{16}	W_5	H_6	6	Q_{12}	**0**	**01**	0	3	Q_{08}
⑨ Q_{15}	W_4	H_5	5						
Q_{28}	W_6	H_8	8	Q_{26}	–	–	1	1	Q_{08}
Q_{27}	W_5	H_7	7	Q_{25}	–	–	1	1	Q_{06}
				Q_{24}	**0**	**1**	0	2	Q_{08}
				Q_{23}	**0**	**10**	0	3	Q_{08}
Q_{38}	W_5	H_8	8	Q_{36}	–	–	1	1	Q_{28}
Q_{37}	W_4	H_7	7	Q_{35}	–	–	2	2	Q_{08}
				Q_{34}	**0**	**11**	0	3	Q_{08}

More nonterminal states					
	L	R	F	N	Q
Q_{48}	**1**	–	0	1	Q_{08}
Q_{47}	**1**	–	0	1	Q_{08}
Q_{46}	**1**	**0**	0	2	Q_{08}
Q_{45}	**1**	**00**	0	3	Q_{08}
Q_{58}	**1**	–	0	1	Q_{08}
⑭ Q_{57}	**1** ⑯	– ⑳	1 ㉑	2 ㉗	Q_{08} ㉓
Q_{56}	**1**	**01**	0	3	Q_{08}
Q_{68}	**1**	**1**	0	2	Q_{08}
Q_{67}	**1**	**10**	0	3	Q_{08}
Q_{78}	**1**	**11**	0	3	Q_{08}

Table 6: Complete implementation of the quasi-arithmetic coding table for $N = 8$. Terminal states are the states that appear in Table 3; nonterminal states are internal states that can be expanded with output. The L and R entires are used only by the encoder, the T and N entires only by the decoder, and all other entires by both. This table and the companion delta array (Table 7) and right-branch array (Table 8) are considerably more compact and faster in operation than the conceptual $N = 8$ quasi-arithmetic coder shown in Table 3.

move down $\Delta = 2$ rows in the code table to Q_{15}, a "terminal state" (one for which no output or interval expansion is possible). But in fact the decision is NOT-FOUND, so ⑩ we use the H entry for state Q_{17}, namely H_7, which indicates that 7 is the high end of the interval $[1, 7)$. ⑪ H_7 is a pointer to one of the four vectors in the right-branch array (Table 8). ⑫ We use $\Delta = 2$ as an index into the H_7 vector, and ⑬ find the next state, Q_{57}. ⑭ We go to state Q_{57} in the code table. It is a nonterminal state, so we perform the output indicated by the L, R, F, and Q entries, which were computed by applying the Witten-Neal-Cleary algorithm to the interval $[5, 7)$.

To do the output, we use a two-byte buffer and two counts (Table 9). We insert new bits into the upper end of the low-order byte, then shift the useful bits into the high-order byte; when the high-order byte is full of useful bits, we output them. Continuing the example, ⑮ suppose that the output buffer contains 6 useful bits, so there is room for 2 more, and that the *pending* count is 2, meaning that the next output bit will be followed by two opposite bits, as in the *bits-to-follow* mechanism of Witten *et al.* [11][5] ⑯ The leading output bit L is **1**, so ⑰ we put **10000000** into the low byte of the buffer (if L had been **0**, we would have put **01111111** into the low byte of the buffer). We then shift left by three bits altogether, one for the leading bit and two for the *pending* bits. Since there was only room for two bits, ⑱ we shift left by two bits, output **01011010**, indicate that space remains for 8 bits, and ⑲ shift left by one more bit. ⑳ The R entry shows that there are no remaining bits. (If there had been, we would have put them into the upper end of the low-order byte of the buffer, then shifted them into the high-order byte.) ㉑ The F entry shows that the *pending* count should be increased by 1. The resulting buffer state is shown at ㉒. Finally, ㉓ the Q entry shows that the next coder state is Q_{08}, indicated at ㉔.

Decoding is more mysterious but slightly easier than encoding. We illustrate it by showing how to decode the decision used in the encoding example. Suppose that the encoded file contains the bytes ... **01011010 01000101 01001000** ..., the first of these bytes being the byte written in the encoding example. Again we maintain a two-byte buffer, shown in Table 10; ㉕ as we begin decoding this decision, all eight bits of the first byte have

[5] Briefly, when the endpoints of the current interval in arithmetic coding are both in the range $[1/4, 3/4)$ but on opposite sides of $1/2$, we know that the next two output bits are **01** or **10**. We do not know what the next bit is, but whatever it is, the *following* bit must be the opposite. So we keep track of this fact, and expand the middle half of the interval. The process can be repeated any number of times.

	W_4	W_5	$W_6$⑥	W_7	W_8
$P = 0$	3	4	5	6	6
$P = 1$	3	4	4	5	6
$P = 2$	3	3	4	5	5
$P = 3$	3	3	4	5	5
$P = 4$	2	3	4	4	5
$P = 5$	2	3	3	4	5
$P = 6$	2	2	3	3	4
$P = 7$	2	2	3	3	4
$P = 8$	2	2	3	3	4
$P = 9$	2	2	3	3	4
$P = 10$	2	2	3	3	3
$P = 11$	2	2	2	3	3
$P = 12$	1	2	2	2	3
$P = 13$	1	2	2	2	3
⑦$P = 14$	1	1	2⑧	2	2
$P = 15$	1	1	1	1	2

Table 7: Delta array. The five vectors, one for each possible terminal state width, are indexed by probability index numbers to find Δ, the size of the right subinterval.

	H_5	H_6	H_7⑪	H_8
$\Delta = 1$	Q_{45}	Q_{56}	Q_{67}	Q_{78}
⑫$\Delta = 2$	Q_{35}	Q_{46}	Q_{57}⑬	Q_{68}
$\Delta = 3$	Q_{25}	Q_{36}	Q_{47}	Q_{58}
$\Delta = 4$	Q_{15}	Q_{26}	Q_{37}	Q_{48}
$\Delta = 5$		Q_{16}	Q_{27}	Q_{38}
$\Delta = 6$			Q_{17}	Q_{28}
$\Delta = 7$				Q_{18}

Table 8: Right branch array. The four vectors, one for each possible value of the high end of a terminal state, are indexed by Δ, the size of the right subinterval, to find a pointer to the next state.

Encoding buffer		Bits left	Pending count
⑮ 10**010110**	00000000	2	2
⑰ 10**010110**	**100**00000	2	3
⑱ **01011010**	**0**0000000	8	1
⑲ 1011010**0**	00000000	7	0
㉒ 1011010**0**	00000000	7	1

Table 9: Encoding example. Useful bits not yet output are shown in bold face type.

Decoding buffer		Bits left
㉕ **11000101**	**01001000**	16
㉘ **00010101**	**001000**00	14
㉙ **10010101**	**001000**00	14

Table 10: Decoding example. Useful bits not yet processed are shown in bold face type.

been consumed, the third byte has been read, and the first bit of the next byte has been changed from **1** to **0**, to account for the pending bits left over from the previous decision. As in the encoder, ④ we are in state Q_{17}, and we find $\Delta = 2$ as in steps ⑤ through ⑧. ㉖ We take the T entry for the current state ($T = 7$, indicating that 7 is the top of the current state) and subtract $\Delta = 2$ to obtain the cutoff value $C = 5$ between the left and right decisions. We shift this value to left-justify it in a byte; since in this coder $N = 8$, three bits of C are significant, so we shift C leftward by 5 bits, giving **10100000**. If the actual value of the high-order byte in the buffer had been less than C, we would have a left (FOUND) branch, but in this case ㉕ the high-order byte **11000101** is greater than (or equal to) the cutoff value, so we have a right (NOT-FOUND) branch. As in steps ⑩ through ⑬, we find the next state to be nonterminal state Q_{57}, indicated at ⑭. ㉗ From the N entry for state Q_{57} we find that 2 bits are to be consumed (corresponding to the output of the leading **1** bit and the incrementing of the *pending* count by 1). ㉘ To consume the two bits, we shift the entire buffer leftward by two bits. (We would have paused to read another byte had the number of useful bits fallen below 9.) Because ㉑ the F entry for state Q_{57} is nonzero, ㉙ we change the value of the high-order bit of the high-order byte, in this case from **0** to **1**. Finally, ㉓ we use the Q entry to find the next state, Q_{08}, indicated at ㉔.

Use of Rice coding. The use of Rice codes to encode the symbol positions is straightforward. The only complication is the difficulty of interleaving the quasi-arithmetic code output and the prefix code output. The bits (or bytes) must be output by the encoder in the order that the decoder will read them. The resulting buffering problem can be solved, but here we sidestep the problem by simply using two separate output files.

4 Analysis of quasi-arithmetic coding

We now show that using quasi-arithmetic coding causes an insignificant increase in the code length compared with pure arithmetic coding. We analyze several cases.

First we assume that we know the success probability p of each event, and we show both how to minimize the average excess code length and how small the excess is. In arithmetic coding we divide the current interval (whose width is W) into subintervals of length L and R, the left subinterval being associated with the *success* event; this gives an effective coding probability $q = L/W$ since the resulting code length is $-\log_2 q$ for the left branch and $-\log_2(1-q)$ for the right. When we encode a binary event with probability p using an effective coding probability q, the average code length $l(p,q)$ is given by

$$l(p,q) = -p \log_2 q - (1-p)\log_2(1-q).$$

If we use exact arithmetic coding, we can subdivide the interval into lengths pW and $(1-p)W$, thus making $q = p$ and giving an average code length equal to the entropy, $-p\log_2 p - (1-p)\log_2(1-p)$; this is optimal.

Consider two probabilities p_1 and p_2 that are adjacent based on the subdivision of an interval of width W; in other words, $p_1 = (W-\Delta_1)/W$, $p_2 = (W-\Delta_2)/W$, and $\Delta_2 = \Delta_1 - 1$. For any probability p between p_1 and p_2, either p_1 or p_2 should be chosen, whichever gives a shorter average code length. There is a cutoff probability p^* for which p_1 and p_2 give the same average code length. We can compute p^* by solving the equation $l(p^*, p_1) = l(p^*, p_2)$, giving

$$p^* = \frac{1}{1 + \dfrac{\log \dfrac{p_2}{p_1}}{\log \dfrac{1-p_1}{1-p_2}}} = \frac{\log \dfrac{\Delta_1}{\Delta_2}}{\log \dfrac{W-\Delta_2}{W-\Delta_1}\dfrac{\Delta_1}{\Delta_2}}. \qquad (1)$$

Clearly we can construct the delta table by computing cutoff probabilities for every pair of adjacent coding probabilities and every possible interval size and then applying them to the count state probabilities. As an example, we compute the value of Δ, the size of the right subinterval, to be used for $\mathsf{F} : \mathsf{NF} = 3 : 1$ (i.e., for $p = {}^3\!/_4$) and $W = 6$. Clearly $\Delta = 1$ or 2, so $p_1 = {}^4\!/_6$ $(\Delta_1 = 2)$ and $p_2 = {}^5\!/_6$ $(\Delta_2 = 1)$. We compute $p^* = \log 2/\log({}^5\!/_2) \approx 0.756$, and choose $\Delta = \Delta_1 = 2$ since $0.667 < 0.750 < 0.756 < 0.833$, i.e., $p_1 < p < p^* < p_2$. This is the entry at ⑧ in Table 7.

Probability p^* is the probability between p_1 and p_2 with the worst average quasi-arithmetic coding performance, both in excess bits per decision and in excess bits relative to optimal compression. (This can be shown by monotonicity arguments.) For a quasi-arithmetic coder with full interval $[0, N)$, the shortest terminal state intervals have size $W = N/4 + 2$; the worst average error occurs for the smallest W and the most extreme probabilities. We bound the absolute and relative average excess code length in the following theorem. (This analysis excludes probabilities less than $1/W$ and greater than $(W-1)/W$, for which the relative excess code length becomes infinite. It is not unusual for probabilities to be very large or small in image compression applications, but in text compression extreme probabilities occur infrequently.)

Theorem 1 *If we construct a quasi-arithmetic coder based on full interval* $[0, N)$, *and use correct probability estimates for probabilities between* $1/N$ *and* $(N-1)/N$, *the number of bits per input symbol by which the average code length obtained by the quasi-arithmetic coder exceeds that of an exact arithmetic coder is at most*

$$\frac{4}{\ln 2}\log_2 \frac{2}{e \ln 2}\frac{1}{N} + O\left(\frac{1}{N^2}\right) \approx \frac{0.497}{N} + O\left(\frac{1}{N^2}\right),$$

and the fraction by which the average code length obtained by the quasi-arithmetic coder exceeds that of an exact arithmetic coder is at most

$$\log_2 \frac{2}{e \ln 2}\frac{1}{\log_2 N} + O\left(\frac{1}{(\log N)^2}\right) \approx \frac{0.0861}{\log_2 N} + O\left(\frac{1}{(\log N)^2}\right).$$

File	Compressed size (bits per input character)				Encoding throughput (thousands of characters per second)			
	Fast PPM		PPMC	*compress*	Fast PPM		PPMC	*compress*
	QA	QA/Rice			QA	QA/Rice		
bib	2.19	2.32	2.12	3.35	23.2	29.0	16.4	111.3
book1	2.51	2.58	2.52	3.46	23.2	30.1	18.5	108.3
book2	2.29	2.41	2.28	3.28	23.5	30.6	18.1	111.1
news	2.78	2.94	2.77	3.86	16.9	23.5	12.6	99.2
paper1	2.62	2.83	2.48	3.77	17.8	24.7	13.6	106.3
paper2	2.51	2.67	2.46	3.52	21.1	26.5	15.2	102.7
progc	2.68	2.92	2.49	3.87	16.9	23.6	12.4	99.0
progl	1.99	2.16	1.87	3.03	24.8	31.7	18.4	119.4
progp	1.96	2.17	1.82	3.11	22.0	31.1	16.5	98.8
trans	1.88	2.09	1.75	3.27	23.7	32.1	18.0	117.1

Table 11: Compression and encoding throughput on the ten text files in the Calgary corpus.

If we let $\overline{p} = (p_1 + p_2)/2$ and note that the maximum value of p in our analysis is $1 - 1/W$, we can expand Equation (1) asymptotically in W to express p^* as

$$p^* = \overline{p} + \frac{1}{6W^2}\,\frac{\overline{p} - {}^{1}\!/_{2}}{\overline{p}\,(1 - \overline{p})} + O(1/W). \tag{2}$$

The $O(\cdot)$ term is $1/W$ because of the effect of the maximum possible value of p. The constant in the $O(1/W)$ term is very small, less than 0.002. We can use Equation (2) to approximate the cutoff probabilities using rational arithmetic; the compression loss introduced by using the approximation $\widetilde{p^*}$ instead of the exact value of p^* is completely negligible, never more than 0.06%. In the example above with $p_1 = {}^{2}\!/_{3}$ and $p_2 = {}^{5}\!/_{6}$, we find that $p^* = \log 2 / \log({}^{5}\!/_{2}) \approx 0.75647$ and $\widetilde{p^*} = 245/324 \approx 0.75617$.

Next we consider a more general case, in which we compare quasi-arithmetic coding with arithmetic coding for a single worst-case event. We assume that both coders use the same estimated probability, but that the estimate need not be right. In this case we find the cutoff probability between p_1 and p_2 for ${}^{1}\!/_{2} \le p_1 < p_2$ by equating the excess code length from using probability p_1 for the more probable event and the excess from using probability p_2 for the less probable event, that is, by solving the equation $-\log_2 p_1 + \log_2 p^* = -\log_2(1 - p_2) + \log_2(1 - p^*)$; this yields

$$p^* = \frac{1}{1 + \dfrac{1 - p_2}{p_1}} = \frac{W - \Delta_1}{W - 1}.$$

The excess code length in this case is just $\log_2(W/(W-1)) \sim 1/W \ln 2$ regardless of the value of Δ_1. We note that the smallest value of W is $N/4 + 2$, and thus we bound the worst-case excess code length in the following theorem.

Theorem 2 *If we construct a quasi-arithmetic coder based on full interval* $[0, N)$*, and use arbitrary probability estimates between* $1/N$ *and* $(N-1)/N$*, the number of bits per input symbol by which the code length obtained by the quasi-arithmetic coder exceeds that of an exact arithmetic coder in the worst case is at most*

$$\log_2 \frac{N+8}{N+4} \sim \frac{4}{N \ln 2} \approx \frac{5.771}{N}.$$

5 Experimental Results

We compare the Fast PPM method with PPMC and with the UNIX *compress* program; the results appear in Table 11. We show results for two versions of Fast PPM: one that uses quasi-arithmetic coding for all binary decisions (QA) and one that uses quasi-arithmetic coding for one decision in each context, then uses Rice coding if necessary to encode the symbol's position in the remainder of the concatenated context list (QA/Rice). For quasi-arithmetic coding use we $N = 32$ and an order-3 coder; the time needed to precompute the tables is not included, since the tables can be compiled into the coder. The PPMC implementation also uses exclusions and an order 3 model. The test data consists of the 10 text files of the Calgary corpus. We see that Fast PPM outcompresses the *compress* program on all text files. Fast PPM with quasi-arithmetic coding gives compression performance comparable to that of PPMC, especially for larger files. We show timing results for encoding on a Sun SPARCstation1GX; decoding times are similar for the PPM methods. We see that Fast PPM, even using quasi-arithmetic coding alone, is always faster than PPMC; the version that uses some Rice coding is nearly twice as fast as PPMC.

6 Conclusion

We have identified several parts of the PPMC text compression method that can be speeded up by the introduction of simplifying approximations. In the Fast PPM method presented here we speed up the modeling phase by eliminating the need for *escape* symbols; since they occur infrequently anyway this does not hurt compression much. We speed up coding by using quasi-arithmetic coding instead of arithmetic coding when we need high-precision predictions, and by using Rice codes to encode the context list positions of low-probability symbols. Quasi-arithmetic coding gives enough precision for practical use as a binary coder and runs much faster than true arithmetic coding; Rice codes waste some code space because of the limitations of their models, but the amount is small because we apply them only to infrequently occurring symbols.

We have presented a detailed example of a quasi-arithmetic coder and its use, and analysis showing that the excess code length introduced is only $O(1/N)$ (in both the average and worst cases) and that the excess relative code length is only $O(1/\log N)$. The analysis is also useful in the construction of the code tables.

Finally, we have shown experimentally that Fast PPM gives compression comparable to that of PPMC, with nearly twice the throughput.

References

[1] R. B. Arps, G. G. Langdon & J. J. Rissanen, "Method for Adaptively Initializing a Source Model for Symbol Encoding," *IBM Technical Disclosure Bulletin* 26 (May 1984), 6292–6294.

[2] T. C. Bell, J. G. Cleary & I. H. Witten, *Text Compression*, Prentice-Hall, Englewood Cliffs, NJ, 1990.

[3] J. G. Cleary & I. H. Witten, "Data Compression Using Adaptive Coding and Partial String Matching," *IEEE Trans. Comm.* COM–32 (Apr. 1984), 396–402.

[4] S. W. Golomb, "Run-Length Encodings," *IEEE Trans. Inform. Theory* IT–12 (July 1966), 399–401.

[5] D. S. Hirschberg & D. A. Lelewer, "Context Modeling for Text Compression," in *Image and Text Compression*, J. A. Storer, ed., Kluwer Academic Publishers, Norwell, MA, 1992, 113–144.

[6] P. G. Howard & J. S. Vitter, "Practical Implementations of Arithmetic Coding," in *Image and Text Compression*, J. A. Storer, ed., Kluwer Academic Publishers, Norwell, MA, 1992, 85–112.

[7] P. G. Howard & J. S. Vitter, "Fast and Efficient Lossless Image Compression," in *Proc. Data Compression Conference*, J. A. Storer & M. Cohn, eds., Snowbird, Utah, Mar. 30-Apr. 1, 1993, 351–360.

[8] A. M. Moffat, "Implementing the PPM Data Compression Scheme," *IEEE Trans. Comm.* COM–38 (Nov. 1990), 1917–1921.

[9] R. F. Rice, "Some Practical Universal Noiseless Coding Techniques," Jet Propulsion Laboratory, JPL Publication 79–22, Pasadena, California, Mar. 1979.

[10] I. H. Witten & T. C. Bell, "The Zero Frequency Problem: Estimating the Probabilities of Novel Events in Adaptive Text Compression," *IEEE Trans. Inform. Theory* IT–37 (July 1991), 1085–1094.

[11] I. H. Witten, R. M. Neal & J. G. Cleary, "Arithmetic Coding for Data Compression," *Comm. ACM* 30 (June 1987), 520–540.