Practical Implementations of Arithmetic Coding

Paul G. Howard and *Jeffrey Scott Vitter*

Brown University
Department of Computer Science

Technical Report No. 92–18
Revised version, April 1992
(Formerly Technical Report No. CS–91–45)

Appears in *Image and Text Compression*,
James A. Storer, ed., Kluwer Academic Publishers, Norwell, MA, 1992, pages 85–112.

A shortened version appears in the proceedings of the
International Conference on Advances in Communication and Control (COMCON 3),
Victoria, British Columbia, Canada, October 16–18, 1991.

Practical Implementations of Arithmetic Coding[1]

Paul G. Howard[2] *Jeffrey Scott Vitter*[3]
Department of Computer Science
Brown University
Providence, R.I. 02912–1910

Abstract

We provide a tutorial on arithmetic coding, showing how it provides nearly optimal data compression and how it can be matched with almost any probabilistic model. We indicate the main disadvantage of arithmetic coding, its slowness, and give the basis of a fast, space-efficient, approximate arithmetic coder with only minimal loss of compression efficiency. Our coder is based on the replacement of arithmetic by table lookups coupled with a new deterministic probability estimation scheme.

Index terms: Data compression, arithmetic coding, adaptive modeling, analysis of algorithms, data structures, low precision arithmetic.

[1]A similar version of this paper appears in *Image and Text Compression*, James A. Storer, ed., Kluwer Academic Publishers, Norwell, MA, 1992, 85–112. A shortened version of this paper appears in the proceedings of the International Conference on Advances in Communication and Control (COMCON 3), Victoria, British Columbia, Canada, October 16–18, 1991.

[2]Support was provided in part by NASA Graduate Student Researchers Program grant NGT–50420 and by a National Science Foundation Presidential Young Investigators Award grant with matching funds from IBM. Additional support was provided by a Universities Space Research Association/CESDIS associate membership.

[3]Support was provided in part by National Science Foundation Presidential Young Investigator Award CCR–9047466 with matching funds from IBM, by NSF research grant CCR–9007851, by Army Research Office grant DAAL03–91–G–0035, and by the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014–91–J–4052 ARPA Order No. 8225. Additional support was provided by a Universities Space Research Association/CESDIS associate membership.

1 Data Compression and Arithmetic Coding

Data can be compressed whenever some data symbols are more likely than others. Shannon [54] showed that for the best possible compression code (in the sense of minimum average code length), the output length contains a contribution of $-\lg p$ bits from the encoding of each symbol whose probability of occurrence is p. If we can provide an accurate model for the probability of occurrence of each possible symbol at every point in a file, we can use arithmetic coding to encode the symbols that actually occur; the number of bits used by arithmetic coding to encode a symbol with probability p is very nearly $-\lg p$, so the encoding is very nearly optimal for the given probability estimates.

In this paper we show by theorems and examples how arithmetic coding achieves its performance. We also point out some of the drawbacks of arithmetic coding in practice, and propose a unified compression system for overcoming them. We begin by attempting to clear up some of the false impressions commonly held about arithmetic coding; it offers some genuine benefits, but it is not the solution to all data compression problems.

The most important advantage of arithmetic coding is its flexibility: it can be used in conjunction with any model that can provide a sequence of event probabilities. This advantage is significant because large compression gains can be obtained only through the use of sophisticated models of the input data. Models used for arithmetic coding may be adaptive, and in fact a number of independent models may be used in succession in coding a single file. This great flexibility results from the sharp separation of the coder from the modeling process [47]. There is a cost associated with this flexibility: the interface between the model and the coder, while simple, places considerable time and space demands on the model's data structures, especially in the case of a multi-symbol input alphabet.

The other important advantage of arithmetic coding is its optimality. Arithmetic coding is optimal in theory and very nearly optimal in practice, in the sense of encoding using minimal average code length. This optimality is often less important than it might seem, since Huffman coding [25] is also very nearly optimal in most cases [8,9, 18,39]. When the probability of some single symbol is close to 1, however, arithmetic coding does give considerably better compression than other methods. The case of highly unbalanced probabilities occurs naturally in bilevel (black and white) image coding, and it can also arise in the decomposition of a multi-symbol alphabet into a sequence of binary choices.

The main disadvantage of arithmetic coding is that it tends to be slow. We shall see that the full precision form of arithmetic coding requires at least one multiplication per event and in some implementations up to two multiplications and two divisions per event. In addition, the model lookup and update operations are slow because of the input requirements of the coder. Both Huffman coding and Ziv-Lempel [59, 60] coding are faster because the model is represented directly in the data structures used for coding. (This reduces the coding efficiency of those methods by narrowing the range of possible models.) Much of the current research in arithmetic coding concerns finding approximations that increase coding speed without compromising compression efficiency. The most common method is to use an approximation to the multiplication operation [10,27,29,43]; in this paper we present an alternative approach using table lookups and approximate probability estimation.

Another disadvantage of arithmetic coding is that it does not in general produce a prefix code. This precludes parallel coding with multiple processors. In addition, the potentially unbounded output delay makes real-time coding problematical in critical applications, but in practice the delay seldom exceeds a few symbols, so this is not a major problem. A minor disadvantage is the need to indicate the end of the file.

One final minor problem is that arithmetic codes have poor error resistance, especially when used with adaptive models [5]. A single bit error in the encoded file causes the decoder's internal state to be in error, making the remainder of the decoded file wrong. In fact this is a drawback of *all* adaptive codes, including Ziv-Lempel codes and adaptive Huffman codes [12,15,18,26,55,56]. In practice, the poor error resistance of adaptive coding is unimportant, since we can simply apply appropriate error correction coding to the encoded file. More complicated solutions appear in [5,20], in which errors are made easy to detect, and upon detection of an error, bits are changed until no errors are detected.

Overview of this paper. In Section 2 we give a tutorial on arithmetic coding. We include an introduction to modeling for text compression. We also restate several important theorems from [22] relating to the optimality of arithmetic coding in theory and in practice.

In Section 3 we present some of our current research into practical ways of improving the speed of arithmetic coding without sacrificing much compression efficiency. The center of this research is a reduced-precision arithmetic coder, supported by efficient data structures for text modeling.

2 Tutorial on Arithmetic Coding

In this section we explain how arithmetic coding works and give implementation details; our treatment is based on that of Witten, Neal, and Cleary [58]. We point out the usefulness of binary arithmetic coding (that is, coding with a 2-symbol alphabet), and discuss the modeling issue, particularly high-order Markov modeling for text compression. Our focus is on encoding, but the decoding process is similar.

2.1 Arithmetic coding and its implementation

Basic algorithm. The algorithm for encoding a file using arithmetic coding works conceptually as follows:

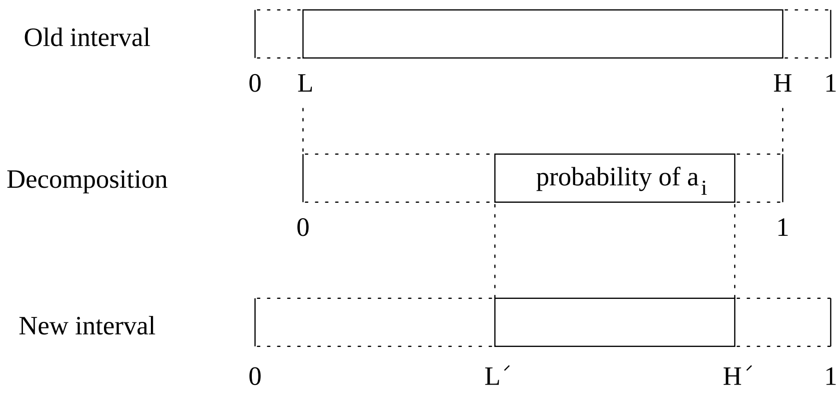

Figure 1: Subdivision of the current interval based on the probability of the input symbol a_i that occurs next.

1. We begin with a "current interval" $[L, H)$ initialized to $[0, 1)$.
2. For each symbol of the file, we perform two steps (see Figure 1):
   (a) We subdivide the current interval into subintervals, one for each possible alphabet symbol. The size of a symbol's subinterval is proportional to the estimated probability that the symbol will be the next symbol in the file, according to the model of the input.
   (b) We select the subinterval corresponding to the symbol that actually occurs next in the file, and make it the new current interval.
3. We output enough bits to distinguish the final current interval from all other possible final intervals.

The length of the final subinterval is clearly equal to the product of the probabilities of the individual symbols, which is the probability p of the particular sequence of symbols in the file. The final step uses almost exactly $-\lg p$ bits to distinguish the file from all other possible files. We need some mechanism to indicate the end of the file, either a special end-of-file symbol coded just once, or some external indication of the file's length.

In step 2, we need to compute only the subinterval corresponding to the symbol a_i that actually occurs. To do this we need two cumulative probabilities, $P_C = \sum_{k=1}^{i-1} p_k$ and $P_N = \sum_{k=1}^{i} p_k$. The new subinterval is $[L + P_C(H - L),\, L + P_N(H - L))$. The need to maintain and supply cumulative probabilities requires the model to have a complicated data structure; Moffat [35] investigates this problem, and concludes for a multi-symbol alphabet that binary search trees are about twice as fast as move-to-front lists.

Example 1: We illustrate a non-adaptive code, encoding the file containing the symbols **bbb** using arbitrary fixed probability estimates $p_{\mathrm{a}} = 0.4$, $p_{\mathrm{b}} = 0.5$, and $p_{\mathsf{EOF}} = 0.1$. Encoding proceeds as follows:

Current Interval	Action	Subintervals a	b	EOF	Input
$[0.000, 1.000)$	Subdivide	$[0.000, 0.400)$	$[0.400, 0.900)$	$[0.900, 1.000)$	**b**
$[0.400, 0.900)$	Subdivide	$[0.400, 0.600)$	$[0.600, 0.850)$	$[0.850, 0.900)$	**b**
$[0.600, 0.850)$	Subdivide	$[0.600, 0.700)$	$[0.700, 0.825)$	$[0.825, 0.850)$	**b**
$[0.700, 0.825)$	Subdivide	$[0.700, 0.750)$	$[0.750, 0.812)$	$[0.812, 0.825)$	EOF
$[0.812, 0.825)$					

The final interval (without rounding) is $[0.8125, 0.825)$, which in binary is approximately $[0.11010\ 00000, 0.11010\ 01100)$. We can uniquely identify this interval by outputting **1101000**. According to the fixed model, the probability p of this particular file is $(0.5)^3(0.1) = 0.0125$ (exactly the size of the final interval) and the code length (in bits) should be $-\lg p = 6.322$. In practice we have to output 7 bits. □

The idea of arithmetic coding originated with Shannon in his seminal 1948 paper on information theory [54]. It was rediscovered by Elias about 15 years later, as briefly mentioned in [1].

Implementation details. The basic implementation of arithmetic coding described above has two major difficulties: the shrinking current interval requires the use of high precision arithmetic, and no output is produced until the entire file has been read. The most straightforward solution to both of these problems is to output each leading bit as soon as it is known, and then to double the length of the current interval so that it reflects only the unknown part of the final interval. Witten, Neal, and Cleary [58] add a clever mechanism for preventing the current interval from shrinking too much when the endpoints are close to $1/2$ but straddle $1/2$. In that case we do not yet know the next output bit, but we do know that whatever it is, the *following* bit will have the opposite value; we merely keep track of that fact, and expand the current interval symmetrically about $1/2$. This follow-on procedure may be repeated any number of times, so the current interval size is always longer than $1/4$.

Mechanisms for incremental transmission and fixed precision arithmetic have been developed through the years by Pasco [40], Rissanen [48], Rubin [52], Rissanen and Langdon [49], Guazzo [19], and Witten, Neal, and Cleary [58]. The bit-stuffing idea of Langdon and others at IBM that limits the propagation of carries in the additions is roughly equivalent to the follow-on procedure described above.

We now describe in detail how the coding and interval expansion work. This process takes place immediately after the selection of the subinterval corresponding to an input symbol.

We repeat the following steps (illustrated schematically in Figure 2) as many times as possible:

a. If the new subinterval is not entirely within one of the intervals $[0, 1/2)$, $[1/4, 3/4)$, or $[1/2, 1)$, we stop iterating and return.

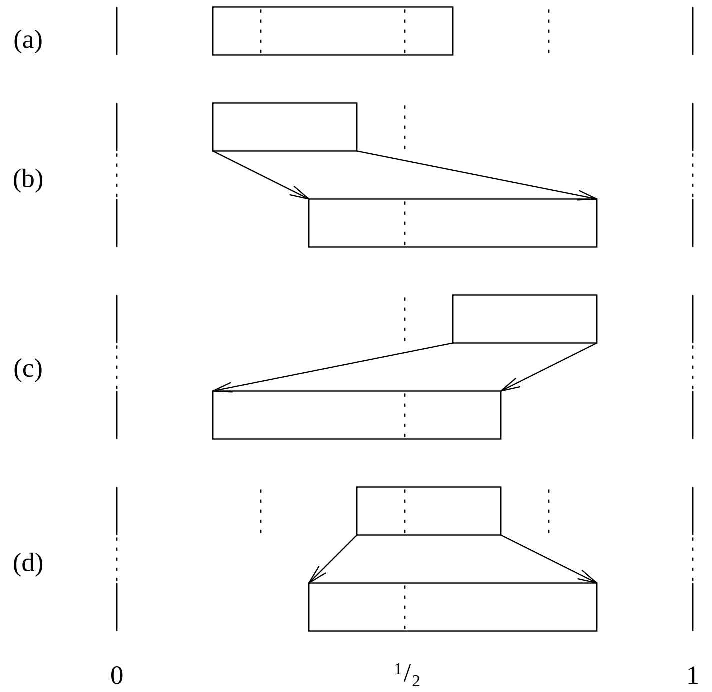

Figure 2: Interval expansion process. (a) No expansion. (b) Interval in $[0, 1/2)$. (c) Interval in $[1/2, 1)$. (d) Interval in $[1/4, 3/4)$ (follow-on case).

b. If the new subinterval lies entirely within $[0, 1/2)$, we output **0** and any **1**s left over from previous symbols; then we double the size of the interval $[0, 1/2)$, expanding toward the right.

c. If the new subinterval lies entirely within $[1/2, 1)$, we output **1** and any **0**s left over from previous symbols; then we double the size of the interval $[1/2, 1)$, expanding toward the left.

d. If the new subinterval lies entirely within $[1/4, 3/4)$, we keep track of this fact for future output; then we double the size of the interval $[1/4, 3/4)$, expanding in both directions away from the midpoint.

Example 2: We show the details of encoding the same file as in Example 1.

Current Interval	Action	Subintervals a	b	EOF	Input
$[0.00, 1.00)$	Subdivide	$[0.00, 0.40)$	$[0.40, 0.90)$	$[0.90, 1.00)$	**b**
$[0.40, 0.90)$	Subdivide	$[0.40, 0.60)$	$[0.60, 0.85)$	$[0.85, 0.90)$	**b**
$[0.60, 0.85)$	Output **1**				
	Expand $[1/2, 1)$				
$[0.20, 0.70)$	Subdivide	$[0.20, 0.40)$	$[0.40, 0.65)$	$[0.65, 0.70)$	**b**
$[0.40, 0.65)$	*follow*				
	Expand $[1/4, 3/4)$				
$[0.30, 0.80)$	Subdivide	$[0.30, 0.50)$	$[0.50, 0.75)$	$[0.75, 0.80)$	EOF
$[0.75, 0.80)$	Output **10**				
	Expand $[1/2, 1)$				
$[0.50, 0.60)$	Output **1**				
	Expand $[1/2, 1)$				
$[0.00, 0.20)$	Output **0**				
	Expand $[0, 1/2)$				
$[0.00, 0.40)$	Output **0**				
	Expand $[0, 1/2)$				
$[0.00, 0.80)$	Output **0**				

The "*follow*" output in the sixth line indicates the follow-on procedure: we keep track of our knowledge that the next output bit will be followed by its opposite; this "opposite" bit is the **0** output in the ninth line. The encoded file is **1101000**, as before. □

Clearly the current interval contains some information about the preceding inputs; this information has not yet been output, so we can think of it as the coder's state. If a is the length of the current interval, the state holds $-\lg a$ bits not yet output. In the basic method (illustrated by Example 1) the state contains *all* the information about the output, since nothing is output until the end. In the implementation illustrated by Example 2, the state always contains fewer than two bits of output information, since the length of the current interval is always more than 1/4. The final state in Example 2 is $[0, 0.8)$, which contains $-\lg 0.8 \approx 0.322$ bits of information.

Use of integer arithmetic. In practice, the arithmetic can be done by storing the current interval in sufficiently long integers rather than in floating point or exact rational numbers. (We can think of Example 2 as using the integer interval $[0, 100)$ by omitting all the decimal points.) We also use integers for the frequency counts used to estimate symbol probabilities. The subdivision process involves selecting non-overlapping intervals (of length at least 1) with lengths approximately proportional to the counts. To encode symbol a_i we need two cumulative counts, $C = \sum_{k=1}^{i-1} c_k$ and $N = \sum_{k=1}^{i} c_k$, and the sum T of all counts, $T = \sum_{k=1}^{n} c_k$. (Here and elsewhere we denote the alphabet size by n.) The new subinterval is $[L + \lfloor \frac{C(H-L)}{T} \rfloor,\ L + \lfloor \frac{N(H-L)}{T} \rfloor)$.

(In this discussion we continue to use half-open intervals as in the real arithmetic case. In implementations [58] it is more convenient to subtract 1 from the right endpoints and use closed intervals. Moffat [36] considers the calculation of cumulative frequency counts for large alphabets.)

Example 3: Suppose that at a certain point in the encoding we have symbol counts $c_{\mathsf{a}} = 4$, $c_{\mathsf{b}} = 5$, and $c_{\mathsf{EOF}} = 1$ and current interval $[25, 89)$ from the full interval $[0, 128)$. Let the next input symbol be **b**. The cumulative counts for **b** are $C = 4$ and $N = 9$, and $T = 10$, so the new interval is $[25 + \lfloor\frac{4(89-25)}{10}\rfloor, 25 + \lfloor\frac{9(89-25)}{10}\rfloor) = [50, 82)$; we then increment the follow-on count and expand the interval once about the midpoint 64, giving $[36, 100)$. It is possible to maintain higher precision, truncating (and adjusting to avoid overlapping subintervals) only when the expansion process is complete; this makes it possible to prove a tight analytical bound on the lost compression caused by the use of integer arithmetic, as we do in [22], restated as Theorem 1 below. In practice this refinement makes the coding more difficult without improving compression. □

Analysis. In [22] we prove a number of theorems about the code lengths of files coded with arithmetic coding. Most of the results involve the use of arithmetic coding in conjunction with various models of the input; these will be discussed in Section 2.3. Here we note two results that apply to implementations of the arithmetic coder. The first shows that using integer arithmetic has negligible effect on code length.

Theorem 1 *If we use integers from the range* $[0, N)$ *and use the high precision algorithm for scaling up the subrange, the code length is provably bounded by* $4/(N \ln 2)$ *bits per input symbol more than the ideal code length for the file.*

For a typical value $N = 65{,}536$, the excess code length is less than 10^{-4} bit per input symbol.

The second result shows that if we indicate end-of-file by encoding a special symbol just once for the entire file, the additional code length is negligible.

Theorem 2 *The use of a special end-of-file symbol when coding a file of length* t *using integers from the range* $[0, N)$ *results in additional code length of less than* $8t/(N \ln 2) + \lg N + 7$ *bits.*

Again the extra code length is negligible, less than 0.01 bit per input symbol for a typical 100,000 byte file.

Since we seldom know the exact probabilities of the process that generated an input file, we would like to know how errors in the estimated probabilities affect the code length. We can estimate the extra code length by a straightforward asymptotic analysis. The average code length L for symbols produced by a given model in a given state is given by

$$L = -\sum_{i=1}^{n} p_i \lg q_i,$$

where p_i is the actual probability of the ith alphabet symbol and q_i is its estimated probability. The optimal average code length for symbols in the state is the entropy of the state, given by

$$H = -\sum_{i=1}^{n} p_i \lg p_i.$$

The excess code length is $E = L - H$; if we let $d_i = q_i - p_i$ and expand asymptotically in d, we obtain

$$E = \sum_{i=1}^{n} \left(\frac{1}{2 \ln 2} \frac{d_i^2}{p_i} + O\left(\frac{d_i^3}{p_i^2}\right)\right). \tag{1}$$

(This corrects a similar derivation in [5], in which the factor of $1/\ln 2$ is omitted.) The vanishing of the linear terms means that small errors in the probabilities used by the coder lead to very small increases in code length. Because of this property, *any* coding method that uses approximately correct probabilities will achieve a code length close to the entropy of the underlying source. We use this fact in Section 3.1 to design a class of fast approximate arithmetic coders with small compression loss.

2.2 Binary arithmetic coding

The preceding discussion and analysis has focused on coding with a multi-symbol alphabet, although in principle it applies to a binary alphabet as well. It is useful to distinguish the two cases since both the coder and the interface to the model are simpler for a binary alphabet. The coding of bilevel images, an important problem with a natural two-symbol alphabet, often produces probabilities close to 1, indicating the use of arithmetic coding to obtain good compression. Historically, much of the arithmetic coding research by Rissanen, Langdon, and others at IBM has focused on bilevel images [29]. The Q-Coder [2,27,33,41,42,43] is a binary arithmetic coder; work by Rissanen and Mohiuddin [50] and Chevion *et al.* [10] extends some of the Q-Coder ideas to multi-symbol alphabets.

In most other text and image compression applications, a multi-symbol alphabet is more natural, but even then we can map the possible symbols to the leaves of a binary tree, and encode an event by traversing the tree and encoding a decision at each internal node. If we do this, the model no longer has to maintain and produce cumulative probabilities; a single probability suffices to encode each decision. Calculating the new current interval is also simplified, since just one endpoint changes after each decision. On the other hand, we now usually have to encode more than one event for each input symbol, and we have a new data structure problem, maintaining the coding trees efficiently without using excessive space. The smallest average number of events coded per input symbol occurs when the tree is a Huffman tree, since such trees have minimum average weighted path length; however, maintaining such trees dynamically is complicated and slow [12,26,55,56]. In Section 3.3 we present a new data structure, the *compressed tree*, suitable for binary encoding of multi-symbol alphabets.

2.3 Modeling for text compression

Arithmetic coding allows us to compress a file as well as possible for a given model of the process that generated the file. To obtain maximum compression of a file, we need both a good model and an efficient way of representing (or learning) the model. (Rissanen calls this principle the *minimum description length* principle; he has investigated it thoroughly from a theoretical point of view [44,45,46].) If we allow two passes over the file, we can identify a suitable model during the first pass, encode it, and use it for optimal coding during the second pass. An alternative approach is to allow the model to adapt to the characteristics of the file during a single pass, in effect learning the model. The adaptive approach has advantages in practice: there is no coding delay and no need to encode the model, since the decoder can maintain the same model as the encoder in a synchronized fashion.

In the following theorem from [22] we compare context-free coding using a two-pass method and a one-pass adaptive method. In the two-pass method, the exact symbol counts are encoded after the first pass; during the second pass each symbol's count is decremented whenever it occurs, so at each point the relative counts reflect the correct symbol probabilities for the remainder of the file (as in [34]). In the one-pass adaptive method, all symbols are given initial counts of 1; we add 1 to a symbol's count whenever it occurs.

Theorem 3 *For all input files, the adaptive code with initial 1-weights gives exactly the same code length as the semi-adaptive decrementing code in which the input model is encoded based on the assumption that all symbol distributions are equally likely.*

Hence we see that use of an adaptive code does not incur any extra overhead, but it does not eliminate the cost of describing the model.

Adaptive models. The simplest adaptive models do not rely on contexts for conditioning probabilities; a symbol's probability is just its relative frequency in the part of the file already coded. (We need a mechanism for encoding a symbol for the first time, when its frequency is 0; the easiest way [58] is to start all symbol counts at 1 instead of 0.) The average code length per input symbol of a file encoded using such a 0-order adaptive model is very close to the 0-order entropy of the file. We shall see that adaptive compression can be improved by taking advantage of locality of reference and especially by using higher order models.

Scaling. One problem with maintaining symbol counts is that the counts can become arbitrarily large, requiring increased precision arithmetic in the coder and more memory to store the counts themselves. By periodically reducing all symbol's counts by the same factor, we can keep the relative frequencies approximately the same while using only a fixed amount of storage for each count. This process is called *scaling*. It allows us to use lower precision arithmetic, possibly hurting compression because of the reduced accuracy of the model. On the other hand, it introduces a *locality of reference* (or *recency*) effect, which often improves compression. We now discuss and quantify the locality effect.

In most text files we find that most of the occurrences of at least some words are clustered in one part of the file. We can take advantage of this locality by assigning more weight to recent occurrences of a symbol in an adaptive model. In practice there are several ways to do this:

- Periodically restarting the model. This often discards too much information to be effective, although Cormack and Horspool find that it gives good results when growing large dynamic Markov models [11].
- Using a sliding window on the text [26]. This requires excessive computational resources.
- Recency rank coding [7,13,53]. This is simple but corresponds to a rather coarse model of recency.
- Exponential aging (giving exponentially increasing weights to successive symbols) [12,38]. This is moderately difficult to implement because of the changing weight increments, although our probability estimation method in Section 3.4 uses an approximate form of this technique.
- Periodic scaling [58]. This is simple to implement, fast and effective in operation, and amenable to analysis. It also has the computationally desirable property of keeping the symbol weights small. In effect, scaling is a practical version of exponential aging.

Analysis of scaling. In [22] we give a precise characterization of the effect of scaling on code length, in terms of an elegant notion we introduce called *weighted entropy*. The weighted entropy of a file at the end of the mth block, denoted by H_m, is the entropy implied by the probability distribution at that time, computed according to the scaling model described above.

We prove the following theorem for a file compressed using arithmetic coding and a zero-order adaptive model with scaling. All counts are halved and rounded up when the sum of the counts reaches $2B$; in effect, we divide the file into b blocks of length B.

Theorem 4 *Let L be the compressed length of a file. Then we have*

$$B\left(\left(\sum_{m=1}^{b} H_m\right) + H_b - H_0\right) - t\,\frac{k}{B} < L < B\left(\left(\sum_{m=1}^{b} H_m\right) + H_b - H_0\right) + t\left(\frac{k}{B}\lg\left(\frac{B}{k_{\min}}\right) + O\left(\frac{k^2}{B^2}\right)\right),$$

Table 1: PPM escape probabilities ($p_{\rm esc}$) and symbol probabilities (p_i). The number of symbols that have occurred j times is denoted by n_j.

	PPMA	PPMB	PPMC	PPMP	PPMX
$p_{\rm esc}$	$\frac{1}{t+1}$	$\frac{k}{t}$	$\frac{k}{t+k}$	$\frac{n_1}{t} - \frac{n_2}{t^2} + \ldots$	$\frac{n_1}{t}$
p_i	$\frac{c_i}{t+1}$	$\frac{c_i-1}{t}$	$\frac{c_i}{t+k}$		

where $H_0 = \lg n$ is the entropy of the initial model, H_m is the (weighted) entropy implied by the scaling model's probability distribution at the end of block m, k is the number of different alphabet symbols that appear in the file, and $k_{\rm min}$ is the smallest number of different symbols that occur in any block.

When scaling is done, we must ensure that no symbol's count becomes 0; an easy way to do this is to round fractional counts to the next higher integer. We show in the following theorem from [22] that this roundup effect is negligible.

Theorem 5 *Rounding counts up to the next higher integer increases the code length for the file by no more than $n/2B$ bits per input symbol.*

When we compare code lengths with and without scaling, we find that the differences are small, both theoretically and in practice.

High order models. The only way to obtain substantial improvements in compression is to use more sophisticated models. For text files, the increased sophistication invariably takes the form of conditioning the symbol probabilities on contexts consisting of one or more symbols of preceding text. (Langdon [28] and Bell, Witten, Cleary, and Moffat [3,4,5] have proven that both Ziv-Lempel coding and the dynamic Markov coding method of Cormack and Horspool [11] can be reduced to finite context models, despite superficial indications to the contrary.)

One significant difficulty with using high-order models is that many contexts do not occur often enough to provide reliable symbol probability estimates. Cleary and Witten deal with this problem with a technique called *Prediction by Partial Matching* (*PPM*). In the PPM methods we maintain models of various context lengths, or *orders*. At each point we use the highest order model in which the symbol has occurred in the current context, with a special *escape* symbol indicating the need to drop to a lower order. Cleary and Witten specify two *ad hoc* methods, called PPMA and PPMB, for computing the probability of the escape symbol. Moffat [37] implements the algorithm and proposes a third method, PPMC, for computing the escape probability: he treats the escape event as a separate symbol; when a symbol occurs for the first time he adds 1 to both the escape count and the new symbol's count. In practice, PPMC compresses better than PPMA and PPMB. PPMP and PPMX appear in [57]; they are based on the assumption that the appearance of symbols for the first time in a file is approximately a Poisson process. See Table 1 for formulas for the probabilities used by the different methods, and see [5] or [6] for a detailed description of the PPM method. In Section 3.5 we indicate two methods that provide improved estimation of the escape probability.

2.4 Other applications of arithmetic coding

Because of its nearly optimal compression performance, arithmetic coding has been proposed as an enhancement to other compression methods and activities related to compression. The output values produced by Ziv-Lempel coding are not uniformly distributed, leading several researchers [21,32,51] to suggest using arithmetic coding to further compress the output. Compression is indeed improved, but at the cost of slowing down the algorithm and increasing its complexity.

Lossless image compression is often performed using predictive coding, and it is often found that the prediction errors follow a Laplace distribution. In [23] we present methods that use tables of the Laplace distribution precomputed for arithmetic coding to obtain excellent compression ratios of grayscale images. The distributions are chosen to guarantee that, for a given variance estimate, the resulting code length exceeds the ideal for the estimate by only a small fixed amount.

Especially when encoding model parameters, it is often necessary to encode arbitrarily large non-negative integers. Witten *et al.* [58] note that arithmetic coding can encode integers according to any given distribution. In the examples in Section 3.1 we show how some encodings of integers found in the literature can be derived as low-precision arithmetic codes.

We point out here that arithmetic coding can also be used to generate random variables from any desired distribution, as well as to produce nearly random bits from the output of any random process. In particular, it is easy to convert random numbers from one base to another, and to convert random bits with an unknown but fixed probability to bits with a probability of 1/2.

3 Fast Arithmetic Coding

In this section we present some of our current research into several aspects of arithmetic coding. We show the construction of a fast, reduced-precision binary arithmetic coder, and indicate a theoretical construct, called the *ϵ-partition*, that can assist in choosing a representative set of probabilities to be used by the coder. We introduce a data structure that we call the *compressed tree* for efficiently representing a multi-symbol alphabet as a binary tree. We give a deterministic algorithm for estimating probabilities of binary events and storing them in 8-bit locations. We give two improved ways of handling the zero-frequency problem (symbols occurring in context for the first time). Finally we show that we can use hashing to obtain fast access of contexts with only a small loss of compression efficiency. All these components can be combined into a fast, space-efficient text coder.

3.1 Reduced-precision arithmetic coding

We have noted earlier that the primary disadvantage of arithmetic coding is its slowness. We have also seen that small errors in probability estimates cause very small increases in code length, so we can expect that by introducing approximations into the arithmetic coding process in a controlled way we can improve coding speed without significantly degrading compression performance. In the Q-Coder work at IBM, the time-consuming multiplications are replaced by additions and shifts, and low-order bits are ignored.

In this section, we take a different approach to approximate arithmetic coding: recalling that the fractional bits characteristic of arithmetic coding are stored as state information in the coder, we reduce the number of possible states, and replace arithmetic operations by table lookups. Here we present a fast, reduced-precision binary arithmetic coder (which we refer to as *quasi-arithmetic coding* in a companion paper [24]) and develop it through a series of examples. It should be noted that the compression is still completely reversible; using reduced precision merely affects the average code length.

The number of possible states (after applying the interval expansion procedure) of an arithmetic coder using the integer interval $[0, N)$ is $3N^2/16$. If we can reduce the number of states to a more manageable level, we can precompute all state transitions and outputs and substitute table lookups for arithmetic in the coder. The obvious way to reduce the number of states is to reduce N. The value of N must be even; for computational convenience we prefer that it be a multiple of 4.

Example 4: The simplest non-trivial coders have $N = 4$, and have only three states. By applying the arithmetic coding algorithm in a straightforward way, we obtain the following coding table. A "*follow*" output indicates application of the follow-on procedure described in Section 2.1.

		0 input		**1** input	
State	Prob{**0**}	Output	Next state	Output	Next state
$[0, 4)$	$0 < p < 1 - \alpha$	**00**	$[0, 4)$	-	$[1, 4)$
	$1 - \alpha \leq p \leq \alpha$	**0**	$[0, 4)$	**1**	$[0, 4)$
	$\alpha < p < 1$	-	$[0, 3)$	**11**	$[0, 4)$
$[0, 3)$	$0 < p < 1/2$	**00**	$[0, 4)$	*follow*	$[0, 4)$
	$1/2 \leq p < 1$	**0**	$[0, 4)$	**10**	$[0, 4)$
$[1, 4)$	$0 < p < 1/2$	**01**	$[0, 4)$	**1**	$[0, 4)$
	$1/2 \leq p < 1$	*follow*	$[0, 4)$	**11**	$[0, 4)$

The value of the cutoff probability α in state $[0,4)$ is clearly between 1/2 and 3/4. If this were an exact coder, the subintervals of length 3 would correspond to $-\lg 3/4 \approx 0.415$ bits of output information stored in the state, and we would choose $\alpha = 1/\lg 3 \approx 0.631$ to minimize the extra code length. But because of the approximate arithmetic, the optimal value of α depends on the distribution of Prob$\{\mathbf{0}\}$; if Prob$\{\mathbf{0}\}$ is uniformly distributed on (0, 1), we find analytically that the excess code length is minimized when $\alpha = (15 - \sqrt{97})/8 \approx 0.644$. Fortunately, the amount of excess code length is not very sensitive to the value of α; in the uniform distribution case any value from about 0.55 to 0.73 gives less than one percent extra code length. $\square$

Arithmetic coding does not mandate any particular assignment of subintervals to input symbols; all that is required is that subinterval lengths be proportional to symbol probabilities and that the decoder make the same assignment as the encoder. In Example 4 we uniformly assigned the left subinterval to symbol **0**. By preventing the longer subinterval from straddling the midpoint whenever possible, we can sometimes obtain a simpler coder that never requires the follow-on procedure; it may also use fewer states.

Example 5: This coder assigns the *right* subinterval to **0** in lines 4 and 7 of Example 4, eliminating the need for using the follow-on procedure; otherwise it is the same as Example 4.

		0 input		**1** input	
State	Prob$\{\mathbf{0}\}$	Output	Next state	Output	Next state
$[0,4)$	$0 < p < 1-\alpha$	**00**	$[0,4)$	-	$[1,4)$
	$1-\alpha \le p \le \alpha$	**0**	$[0,4)$	**1**	$[0,4)$
	$\alpha < p < 1$	-	$[0,3)$	**11**	$[0,4)$
$[0,3)$	$0 < p < 1/2$	**10**	$[0,4)$	**0**	$[0,4)$
	$1/2 \le p < 1$	**0**	$[0,4)$	**10**	$[0,4)$
$[1,4)$	$0 < p < 1/2$	**01**	$[0,4)$	**1**	$[0,4)$
	$1/2 \le p < 1$	**1**	$[0,4)$	**01**	$[0,4)$

$\square$

Langdon and Rissanen [29] suggest identifying the symbols as the more probable symbol (MPS) and less probable symbol (LPS) rather than as **1** and **0**. By doing this we can often combine transitions and eliminate states.

Example 6: We modify Example 5 to use the MPS/LPS idea. We are able to reduce the coder to just two states.

		LPS input		MPS input	
State	Prob{MPS}	Output	Next state	Output	Next state
$[0, 4)$	$1/2 \le p \le \alpha$	**0**	$[0, 4)$	**1**	$[0, 4)$
	$\alpha < p < 1$	**00**	$[0, 4)$	-	$[1, 4)$
$[1, 4)$	$1/2 \le p < 1$	**01**	$[0, 4)$	**1**	$[0, 4)$

□

Another way of simplifying an arithmetic coder is to allow only a subset of the possible interval subdivisions. Using integer arithmetic has the effect of making the symbol probabilities approximate, especially as the integer range is made smaller; limiting the number of subdivisions simply makes them even less precise. Since the main benefit of arithmetic coding is its ability to code efficiently when probabilities are close to 1, we usually want to allow at least some pairs of unequal probabilities.

Example 7: If we know that one symbol occurs considerably more often than the other, we can eliminate the transitions in Example 6 for approximately equal probabilities. This makes it unnecessary for the coder to decide which transition pair to use in the $[0, 4)$ state, and gives a very simple reduced-precision arithmetic coder.

	LPS input		MPS input	
State	Output	Next state	Output	Next state
$[0, 4)$	**00**	$[0, 4)$	-	$[1, 4)$
$[1, 4)$	**01**	$[0, 4)$	**1**	$[0, 4)$

This simple code is quite useful, providing almost a 50 percent improvement on the unary code for representing non-negative integers. To encode n in unary, we output n **1**s and a **0**. Using the code just derived, we re-encode the unary coding, treating **1** as the MPS. The resulting code consists of $\lfloor n/2 \rfloor$ **1**s, followed by **00** if n is even and **01** if n is odd. We can do even better with slightly more complex codes, as we shall see in examples that follow. □

We now introduce the *maximally unbalanced subdivision* and show how it can be used to obtain excellent compression when Prob{MPS} ≈ 1. Suppose the current interval is $[L, H)$. If Prob{MPS} is very high we can subdivide the interval at $L + 1$ or $H - 1$, indicating Prob{LPS} $= 1/(H - L)$ and Prob{MPS} $= 1 - 1/(H - L)$. Since the length of the current interval $H - L$ is always more than $N/4$, such a subdivision always indicates a Prob{MPS} of more than $1 - 4/N$. By choosing a large value of N and always including the maximally unbalanced subdivision in our coder, we ensure that very likely symbols can always be given an appropriately high probability.

Example 8: Let $N = 8$ and let the MPS always be **1**. We obtain the following four-state code if we allow only the maximally unbalanced subdivision in each state.

State	**0** (LPS) input Output	Next state	**1** (MPS) input Output	Next state
$[0,8)$	**000**	$[0,8)$	-	$[1,8)$
$[1,8)$	**001**	$[0,8)$	-	$[2,8)$
$[2,8)$	**010**	$[0,8)$	-	$[3,8)$
$[3,8)$	**011**	$[0,8)$	**1**	$[0,8)$

We can use this code to re-encode unary-coded non-negative integers with $\lfloor n/4 \rfloor + 3$ bits. In effect, we represent n in the form $4a+b$; we encode a in unary, then use two bits to encode b in binary. □

Whenever the current interval coincides with the full interval, we can switch to a different code.

Example 9: We can derive the Elias code for the positive integers [14] by using the maximally unbalanced subdivision technique of Example 8 and by doubling the full integer range whenever we see enough **1**s to output a bit and expand the current interval so that it coincides with the full range. This coder has an infinite number of states; no state is visited more than once. We use the notation $[L,H)/M$ to indicate the subinterval $[L,H)$ selected from the range $[0,M)$.

State	**0** (LPS) input Output	Next state	**1** (MPS) input Output	Next state
$[0,2)/2$	**0**	STOP	**1**	$[0,4)/4$
$[0,4)/4$	**00**	STOP	-	$[1,4)/4$
$[1,4)/4$	**01**	STOP	**1**	$[0,8)/8$
$[0,8)/8$	**000**	STOP	-	$[1,8)/8$
$[1,8)/8$	**001**	STOP	-	$[2,8)/8$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

This code corresponds to encoding positive integers as follows:

n	Code
1	**0**
2	**100**
3	**101**
4	**11000**
5	**11001**
$\vdots$	$\vdots$

In effect we represent n in the form $2^a + b$; we encode a in unary, then use a bits to encode b in binary. This is essentially the Elias code; it requires $\lfloor 2 \lg n \rfloor + 1$ bits to encode n. □

If we design a coder with more states, we obtain a more fine-grained set of probabilities.

Example 10: We show a six-state coder, obtained by letting $N = 8$ and allowing all possible subdivisions. We indicate only the center probability for each range; in practice any reasonable division will give good results. Output symbol f indicates application of the follow-on procedure.

State	Approximate Prob{MPS}	LPS input Output	LPS input Next state	MPS input Output	MPS input Next state
$[0,8)$	$1/2$	**1**	$[0,8)$	**0**	$[0,8)$
	$5/8$	**1**	$[2,8)$	-	$[0,5)$
	$3/4$	**11**	$[0,8)$	-	$[0,6)$
	$7/8$	**111**	$[0,8)$	-	$[0,7)$
$[0,7)$	$4/7$	**1**	$[0,6)$	**0**	$[0,8)$
	$5/7$	**1**f	$[0,8)$	-	$[0,5)$
	$6/7$	**110**	$[0,8)$	-	$[0,6)$
$[0,6)$	$1/2$	f	$[2,8)$	**0**	$[0,6)$
	$2/3$	**10**	$[0,8)$	**0**	$[0,8)$
	$5/6$	**101**	$[0,8)$	-	$[0,5)$
$[2,8)$	$1/2$	f	$[0,6)$	**1**	$[2,8)$
	$2/3$	**01**	$[0,8)$	**1**	$[0,8)$
	$5/6$	**010**	$[0,8)$	-	$[3,8)$
$[0,5)$	$3/5$	ff	$[0,8)$	**0**	$[0,6)$
	$4/5$	**100**	$[0,8)$	**0**	$[0,8)$
$[3,8)$	$3/5$	ff	$[0,8)$	**1**	$[2,8)$
	$4/5$	**011**	$[0,8)$	**1**	$[0,8)$

This coder is easily programmed and extremely fast. Its only shortcoming is that on average high-probability symbols require 1/4 bit (corresponding to Prob{MPS} $= 2^{-1/4} \approx 0.841$) no matter how high the actual probability is. □

Design of a class of reduced-precision coders. We now present a very flexible yet simple coder design incorporating most of the features just discussed. We choose N to be any power of 2. All states in the coder are of the form $[k, N)$, so the number of states is only $N/2$. (Intervals with $k \geq N/2$ will produce output, and the interval will be expanded.) In every state $[k, N)$ we include the maximally unbalanced subdivision (at $k+1$), which corresponds to values of Prob{MPS} between $(N-2)/N$ and $(N-1)/N$. We include a nearly balanced subdivision so that we will not lose efficiency when Prob{MPS} $\approx 1/2$. In addition, we locate other subdivision points such that the subinterval expansion that follows each input symbol leaves the coder in a state of the form $[k, N)$, and we choose one or more of them to correspond to intermediate values of Prob{MPS}. For simplicity we denote state $[k, N)$ by k.

We always allow the interval $[k, N)$ to be divided at $k+1$; if the LPS occurs we output the $\lg N$ bits of k and move to state 0, while if the MPS occurs we simply move to state $k+1$, then if the new state is $N/2$ we output a **1** and move to state 0. The other permitted subdivisions are given in the following table. In some cases additional output and expansion may be possible. It may not be necessary to include all subdivisions in the coder.

Range of states k	Subdivision LPS	Subdivision MPS	LPS input Output	LPS input Next State	MPS input Output	MPS input Next State
$[0, \frac{N}{2})$	$[k, \frac{N}{2})$	$[\frac{N}{2}, N)$	**0**	$2k$	**1**	0
$[0, \frac{N}{4})$	$[k, \frac{N}{4})$	$[\frac{N}{4}, N)$	**00**	$4k$	-	$\frac{N}{4}$
$[\frac{N}{8}, \frac{N}{4})$	$[k, \frac{3N}{8})$	$[\frac{3N}{8}, N)$	**0**f	$4k - \frac{N}{2}$	-	$\frac{3N}{8}$
$[\frac{N}{4}, \frac{3N}{8})$	$[k, \frac{3N}{8})$	$[\frac{3N}{8}, N)$	**010**	$8k - 2N$	-	$\frac{3N}{8}$
$[\frac{3N}{8}, \frac{N}{2})$	$[k, \frac{5N}{8})$	$[\frac{5N}{8}, N)$	ff	$4k - \frac{3N}{2}$	**1**	$\frac{N}{4}$
$[\frac{7N}{16}, \frac{N}{2})$	$[k, \frac{9N}{16})$	$[\frac{9N}{16}, N)$	fff	$8k - \frac{7N}{2}$	**1**	$\frac{N}{8}$
$[\frac{N}{4}, \frac{N}{2})$	$[\frac{3N}{4}, N)$	$[k, \frac{3N}{4})$	**11**	0	f	$2k - \frac{N}{2}$

For example, the fifth line indicates that for all states k for which $3N/8 \leq k < N/2$, we may subdivide the interval at $5N/8$. If the LPS occurs, we perform the follow-on procedure twice, which leaves us with the interval $[4k - 3N/2, N)$; otherwise we output a **1** and expand the interval to $[N/4, N)$.

A coder constructed using this procedure will have a small number of states, but in every state it will allow us to use estimates of Prob{MPS} near 1, near 1/2, and in between. Thus we can choose a large N so that highly probable events require negligible code length, while keeping the number of states small enough to allow table lookups rather than arithmetic.

3.2 ϵ-partitions and ρ-partitions

In Section 3.1 we have shown that is is possible to design a binary arithmetic coder that admits only a small number of possible probabilities. In this section we give a theoretical basis for selecting the probabilities. Often there are practical considerations limiting our choices, but we can show that it is reasonable to expect that choosing only a few probabilities will give close to optimal compression.

For a binary alphabet, we can use Equation (1) to compute $E(p, q)$, the extra code length resulting from using estimates q and $1 - q$ for actual probabilities p and $1 - p$, respectively. For any desired maximum excess code length ϵ, we can partition the space of possible probabilities to guarantee that the use of approximate probabilities will never add more than ϵ to the code length of any event. We select partitioning probabilities $P_0, P_1, \ldots$ and estimated probabilities $Q_0, Q_1, \ldots$. Each probability Q_i is used to encode all events whose probability p is in the range $P_i < p \leq P_{i+1}$. We compute the partition, which we call an *ϵ-partition*, as follows:

1. Set $i := 0$ and $Q_0 := 1/2$.
2. Find the value of P_{i+1} (greater than Q_i) such that $E(P_{i+1}, Q_i) = \epsilon$. We will use Q_i as the estimated probability for all probabilities p such that $Q_i < p \leq P_{i+1}$.
3. Find the value of Q_{i+1} (greater than P_{i+1}) such that $E(P_{i+1}, Q_{i+1}) = \epsilon$. After we compute P_{i+2} in step 2 of the next iteration, we will use Q_{i+1} as the estimate for all probabilities p such that $P_{i+1} < p \leq P_{i+2}$.

We increment i and repeat steps 2 and 3 until P_{i+1} or Q_{i+1} reaches 1. The values for $p < 1/2$ are symmetrical with those for $p > 1/2$.

Example 11: We show the ϵ-partition for $\epsilon = 0.05$ bit per binary input symbol.

Range of actual probabilities	Probability to use
$[0.0000, 0.0130)$	0.0003
$[0.0130, 0.1427)$	0.0676
$[0.1427, 0.3691)$	0.2501
$[0.3691, 0.6309)$	0.5000
$[0.6309, 0.8579)$	0.7499
$[0.8579, 0.9870)$	0.9324
$[0.9870, 1.0000)$	0.9997

Thus by using only 7 probabilities we can guarantee that the excess code length does not exceed 0.05 bit for each binary decision coded. □

We might wish to limit the *relative* error so that the code length can never exceed the optimal by more than a factor of $1 + \rho$. We can begin to compute these *ρ-partitions* using a procedure similar to that for ϵ-partitions, but unfortunately the process does not terminate, since ρ-partitions are not finite. As P approaches 1, the optimal average code length grows very small, so to obtain a small relative loss Q must be very close to P. Nevertheless, we can obtain a partial ρ-partition.

Example 12: We show part of the ρ-partition for $\rho = 0.05$; the maximum relative error is 5 percent.

Range of actual probabilities	Probability to use
$\vdots$	$\vdots$
$[0.0033, 0.0154)$	0.0069
$[0.0154, 0.0573)$	0.0291
$[0.0573, 0.1670)$	0.0982
$[0.1670, 0.3722)$	0.2555
$[0.3722, 0.6278)$	0.5000
$[0.6278, 0.8330)$	0.7445
$[0.8330, 0.9427)$	0.9018
$[0.9427, 0.9846)$	0.9709
$[0.9846, 0.9967)$	0.9931
$\vdots$	$\vdots$

$\square$

In practice we will use an approximation to an ϵ-partition or a ρ-partition for values of Prob{MPS} up to the maximum probability representable by our coder.

3.3 Compressed trees

To use the reduced-precision arithmetic coder described in Section 3.1 for an n-symbol alphabet, we need an efficient data structure to map each of n symbols to a sequence of binary choices. We might consider Huffman trees, since they minimize the average number of binary events encoded per input symbol; however, a great deal of effort is required to keep the probabilities on all branches near 1/2. For arithmetic coding maintaining this balance condition is unnecessary and wastes time.

In this section we present the *compressed tree*, a space-efficient data structure based on the complete binary tree. Because arithmetic coding allows us to obtain nearly optimal compression of binary events even when the two probabilities are unequal, we are free to represent the probability distribution of an n-symbol alphabet by a complete binary tree with a probability at each internal node. The tree can be flattened (linearized) by breadth-first traversal, and we can save space by storing only one probability at each internal node, say, the probability of taking the left branch. (This probability can be stored to sufficient precision in just one byte, as we shall see in Section 3.4.)

In high-order text models, many longer contexts occur only a few times, and only a few different alphabet symbols occur in each context. In such cases even the linear representation is wasteful of space, requiring $n - 1$ nodes regardless of the number of alphabet symbols that actually occur. Including pointers in the nodes would at

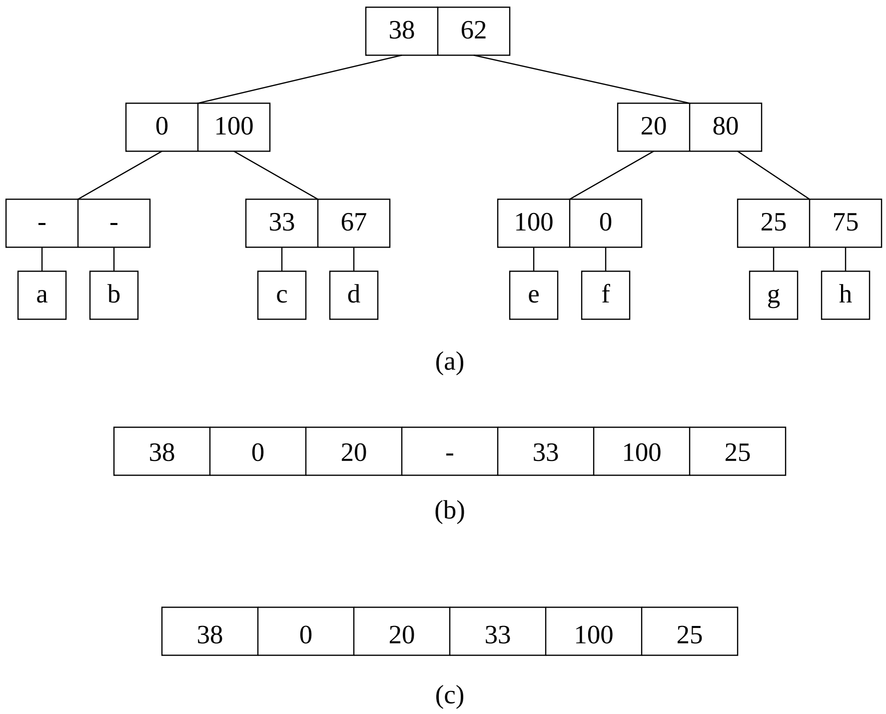

Figure 3: Steps in the development of a compressed tree. (a) Complete binary tree. (b) Linear representation. (c) Compressed tree.

least double their size. In the compressed tree we collapse the breath-first linear representation of the complete binary tree by omitting nodes with zero probability. If k different symbols have non-zero probability, the compressed tree representation requires at most $k \lg(2n/k) - 1$ nodes.

Example 13: Suppose we have the following probability distribution for an 8-symbol alphabet:

Symbol	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**h**
Probability	0	0	$1/8$	$1/4$	$1/8$	0	$1/8$	$3/8$

We can represent this distribution by the tree in Figure 3(a), rounding probabilities and expressing them as multiples of 0.01. We show the linear representation in Figure 3(b) and the compressed tree representation in Figure 3(c). □

Traversing the compressed tree is mainly a matter of keeping track of omitted nodes. We do not have to process each node of the tree: for the first $\lg n - 2$ levels we have to process each node; but when we reach the desired node in the next-to-lowest level we have enough information to directly index the desired node of the lowest level. The operations are very simple, involving only one test and one or two increment operations at each node, plus a few extra operations at each level. Including the capability of adding new symbols to the tree makes the algorithm only slightly more complicated.

3.4 Representing and estimating probabilities

In our binary coded representation of each context we wish to use only one byte for each probability, and we need the probability to limited precision. Therefore, we will represent the probability at a node as a state in a finite state automaton with about 256 states. Each state indicates a probability, and some of the states also indicate the size of the sample used to estimate the probability.

We need a method for estimating the probability at each node of the binary tree. Leighton and Rivest [30] and Pennebaker and Mitchell [41] describe probabilistic methods. Their estimators are also finite state automata, with each state corresponding to a probability. When a new symbol occurs, a transition to another state may occur, the probability of the transition depending on the current state and the new symbol. Generally, the transition probability is higher when the LPS occurs. In [30] transitions occur only between adjacent states. In [41] the LPS always causes a transition, possibly to a non-adjacent state; a transition after the MPS, when one occurs, is always to an adjacent state.

We give a deterministic estimator based on the same idea. In our estimator each input symbol causes a transition (unless the MPS occurs when the estimated probability is already at its maximum value). The probabilities represented by the states are so close together that transitions often occur between non-adjacent states. The transitions are selected so that we compute the new probability p_{new} of the left branch by

$$p_{\mathrm{new}} \approx \begin{cases} f\,p_{\mathrm{old}} + (1-f) & \text{if the left branch was taken} \\ f\,p_{\mathrm{old}} & \text{if the right branch was taken,} \end{cases}$$

where f is a smoothing factor. This corresponds to exponential aging; hence the probability estimate can track changing probabilities and benefit from locality of reference, as discussed in Section 2.3.

In designing a probability estimator of this type we must choose both the scaling factor f and the set of probabilities represented by the states. We should be guided by the requirements of the coder and by our lack of *a priori* knowledge of the process generating the sequence of branches.

First we note that when the number of occurrences is small, our estimates cannot be very accurate. Laplace's law of succession, which gives the estimate

$$p = \frac{c+1}{t+2} \tag{2}$$

after c successes in t trials, offers a good balance between using all available information and allowing for random variation in the data; in effect, it gives the Bayesian estimate assuming a uniform *a priori* distribution for the true underlying probability P.

We recall that for values of P near $1/2$ we do not require a very accurate estimate, since any value will give about the same code length; hence we do not need many states in this probability region. When P is closer to 1, we would like our estimate to be more accurate, to allow the arithmetic coder to give near-optimal compression, so we assign states more densely for larger P. Unfortunately, in this case estimation by any means is difficult, because occurrences of the LPS are so infrequent. We also note that the underlying probability of any branch in the coding tree may change at any time, and we would like our estimate to adapt accordingly.

To handle the small-sample cases, we reserve a number of states simply to count occurrences when t is small, using Equation (2) to estimate the probabilities. We do the same for larger values of t when c is 0, 1, $t-1$, or t, to provide fast convergence to extreme values of P.

We can show that if the underlying probability P does not change, the expected value of the estimate p_k after k events is given by

$$E(p_k) = P + (p_0 - P)f^k,$$

which converges to P for all f, $0 \le f < 1$. The rapid convergence of $E(p_k)$ when $f = 0$ is misleading, since in that case the estimate is always 0 or 1, depending only on the preceding event. The expected value is clearly P, but the estimator is useless. A value of f near 1 provides resistance to random fluctuations in the input, but the estimate converges slowly, both initially and when the underlying P changes. A careful choice of f would depend on a detailed analysis like that performed by Flajolet for the related problem of approximate counting [16,17]. We make a more pragmatic decision. We know that periodic scaling is an approximation to exponential aging and we can show that a scaling factor of f corresponds to a scaling block size B of approximately $f \ln 2/(1-f)$. Since $B = 16$ works well for scaling [58], we choose $f = 0.96$.

3.5 Improved modeling for text compression

To obtain good, fast text compression, we wish to use the multi-symbol extension of the reduced-precision arithmetic coder in conjunction with a good model. The PPM idea described in Section 2.3 has proven effective, but the *ad hoc* nature of the escape probability calculation is somewhat annoying. In this section we present yet another *ad hoc* method, which we call PPMD, and also a more complicated but more principled approach to the problem.

PPMD. Moffat's PPMC method [37] is widely considered to be the best method of estimating escape probabilities. In PPMC, each symbol's weight in a context is taken to be number of times it has occurred so far in the context. The escape "event,"

Table 2: Comparsion of PPMC and PPMD. Compression figures are in bits per input symbol.

File	Text?	PPMC	PPMD	Improvement using PPMD
bib	Yes	2.11	2.09	0.02
book1	Yes	2.65	2.63	0.02
book2	Yes	2.37	2.35	0.02
news	Yes	2.91	2.90	0.01
paper1	Yes	2.48	2.46	0.02
paper2	Yes	2.45	2.42	0.03
paper3	Yes	2.70	2.68	0.02
paper4	Yes	2.93	2.91	0.02
paper5	Yes	3.01	3.00	0.01
paper6	Yes	2.52	2.50	0.02
progc	Yes	2.48	2.47	0.01
progl	Yes	1.87	1.85	0.02
progp	Yes	1.82	1.80	0.02
geo	No	5.11	5.10	0.01
obj1	No	3.68	3.70	-0.02
obj2	No	2.61	2.61	0.00
pic	No	0.95	0.94	0.01
trans	No	1.74	1.72	0.02

that is, the occurrence of a symbol for the first time in the context, is also treated as a "symbol," with its own count. When a letter occurs for the first time, its weight becomes 1; the escape count is incremented by 1, so the total weight increases by 2. At all other times the total weight increases by 1.

We have developed a new method, which we call PPMD, which is similar to PPMC except that it makes the treatment of new symbols more consistent by adding 1/2 instead of 1 to both the escape count and the new symbol's count when a new symbol occurs; hence the total weight always increases by 1. We have compared PPMC and PPMD on the Bell-Cleary-Witten corpus [5] (including the four papers not described in the book). Table 2 shows that for text files PPMD compresses consistently about 0.02 bit per character better than PPMC. The compression results for PPMC differ from those reported in [5] because of implementation differences; we used versions of PPMC and PPMD that were identical except for the escape probability calculations. PPMD has the added advantage of making analysis more tractable by making the code length independent of the appearance order of symbols in the context.

Indirect probability estimation. Often we are faced with a situation where we have no theoretical basis for estimating the probability of an event, but where we know the factors that affect the probability. In such cases a logical and effective approach is to create conditioning classes based on the values of the factors, and to estimate the probability adaptively for each class. In the PPM method, we know that the number of occurrences of a state (t) and the number of different alphabet symbols that have occurred (k) are the factors affecting p_{esc}. We have done experiments, using all combinations of t and k as the conditioning classes (except that we group together all values of t greater than 48 and all values of k greater than 18). In our experiments we use a third-order model; when a symbol has not occurred previously in its context of length 3, we simply use 8 bits to indicate the ASCII value of the symbol. (The idea of skipping some shorter contexts for speed, space, and simplicity appears also in [31].) Even with this simplistic way of dropping to shorter contexts, the improved estimation of p_{esc} gives slightly better overall compression than PPMC for *book1*, the longest file in the Bell-Cleary-Witten corpus. We expect that using indirect probability estimation in conjunction with the full multi-order PPM mechanism will yield substantially improved compression.

3.6 Hashed high-order Markov models.

For finding contexts in the PPM method, Moffat [37] and Bell *et al.* [5] give complicated data structures called *backward trees* and *vine pointers*. For fast access and minimal memory usage we propose single hashing without collision resolution. One might expect that using the same bucket for accumulating statistics from unrelated contexts would significantly degrade compression performance, but we can show that often this is not the case.

Even in the worst case, when the symbols from the k colliding contexts in bucket b are mutually disjoint, the additional code length is only $H_b = H(p_1, p_2, p_3, \ldots, p_k)$, the entropy of the ensemble of probabilities of occurrence of the contexts. We show this by conceptually dividing the bucket into disjoint subtrees corresponding to the various contexts, and noting that the cost of identifying an individual symbol is just $L_C = -\lg p_i$, the cost of identifying the context that occurred, plus L_S, the cost of identifying the symbol in its own context. Hence the extra cost is just L_C, and the average extra cost is $\sum_{i=1}^{k} -p_i \lg p_i = H_b$. The maximum value of H_b is $\lg k$, so in buckets that contain data from only two contexts, the extra code length is at most 1 bit per input symbol.

In fact, when the number of colliding contexts in a bucket is large enough that H_b is significant, the symbols in the bucket, representing a combination of a number of contexts, will be a microcosm of the entire file; the bucket's average code length will approximately equal the 0-order entropy of the file. Lelewer and Hirschberg [31] apply hashing with collision resolution in a similar high-order scheme.

4 Conclusion

We have shown the details of an implementation of arithmetic coding and have pointed out its advantages (flexibility and near-optimality) and its main disadvantage (slowness). We have developed a fast coder, based on reduced-precision arithmetic coding, which gives only minimal loss of compression efficiency; we can use the concept of ϵ-partitions to find the probabilities to include in the coder to keep the compression loss small. In a companion paper [24], in which we refer to this fast coding method as *quasi-arithmetic coding*, we give implementation details and performance analysis for both binary and multi-symbol alphabets. We prove analytically that the loss in compression efficiency compared with exact arithmetic coding is negligible.

We introduce the compressed tree, a new data structure for efficiently representing a multi-symbol alphabet by a series of binary choices. Our new deterministic probability estimation scheme allows fast updating of the model stored in the compressed tree using only one byte for each node; the model can provide the reduced-precision coder with the probabilities it needs. Choosing one of our two new methods for computing the escape probability enables us to use the highly effective PPM algorithm, and use of a hashed Markov model keeps space and time requirements manageable even for a high-order model.

References

[1] N. Abramson, *Information Theory and Coding*, McGraw-Hill, New York, NY, 1963.

[2] R. B. Arps, T. K. Truong, D. J. Lu, R. C. Pasco & T. D. Friedman, "A Multi-Purpose VLSI Chip for Adaptive Data Compression of Bilevel Images," *IBM J. Res. Develop.* 32 (Nov. 1988), 775–795.

[3] T. Bell, "A Unifying Theory and Improvements for Existing Approaches to Text Compression," Univ. of Canterbury, Ph.D. Thesis, 1986.

[4] T. Bell & A. M. Moffat, "A Note on the DMC Data Compression Scheme," *Computer Journal* 32 (1989), 16–20.

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

[6] T. C. Bell, I. H. Witten & J. G. Cleary, "Modeling for Text Compression," *Comput. Surveys* 21 (Dec. 1989), 557–591.

[7] J. L. Bentley, D. D. Sleator, R. E. Tarjan & V. K. Wei, "A Locally Adaptive Data Compression Scheme," *Comm. ACM* 29 (Apr. 1986), 320–330.

[8] A. C. Blumer & R. J. McEliece, "The Rényi Redundancy of Generalized Huffman Codes," *IEEE Trans. Inform. Theory* IT–34 (Sept. 1988), 1242–1249.

[9] R. M. Capocelli, R. Giancarlo & I. J. Taneja, "Bounds on the Redundancy of Huffman Codes," *IEEE Trans. Inform. Theory* IT–32 (Nov. 1986), 854–857.

[10] D. Chevion, E. D. Karnin & E. Walach, "High Efficiency, Multiplication Free Approximation of Arithmetic Coding," in *Proc. Data Compression Conference*, J. A. Storer & J. H. Reif, eds., Snowbird, Utah, Apr. 8–11, 1991, 43–52.

[11] G. V. Cormack & R. N. Horspool, "Data Compression Using Dynamic Markov Modelling," *Computer Journal* 30 (Dec. 1987), 541–550.

[12] G. V. Cormack & R. N. Horspool, "Algorithms for Adaptive Huffman Codes," *Inform. Process. Lett.* 18 (Mar. 1984), 159–165.

[13] P. Elias, "Interval and Recency Rank Source Coding: Two On-line Adaptive Variable Length Schemes," *IEEE Trans. Inform. Theory* IT–33 (Jan. 1987), 3–10.

[14] P. Elias, "Universal Codeword Sets and Representations of Integers," *IEEE Trans. Inform. Theory* IT–21 (Mar. 1975), 194–203.

[15] N. Faller, "An Adaptive System for Data Compression," Record of the 7th Asilomar Conference on Circuits, Systems, and Computers, 1973.

[16] Ph. Flajolet, "Approximate Counting: a Detailed Analysis," *BIT* 25 (1985), 113.

[17] Ph. Flajolet & G. N. N. Martin, "Probabilistic Counting Algorithms for Data Base Applications," INRIA, Rapport de Recherche No. 313, June 1984.

[18] R. G. Gallager, "Variations on a Theme by Huffman," *IEEE Trans. Inform. Theory* IT–24 (Nov. 1978), 668–674.

[19] M. Guazzo, "A General Minimum-Redundancy Source-Coding Algorithm," *IEEE Trans. Inform. Theory* IT–26 (Jan. 1980), 15–25.

[20] M. E. Hellman, "Joint Source and Channel Encoding," Proc. Seventh Hawaii International Conf. System Sci., 1974.

[21] R. N. Horspool, "Improving LZW," in *Proc. Data Compression Conference*, J. A. Storer & J. H. Reif, eds., Snowbird, Utah, Apr. 8–11, 1991, 332–341.

[22] P. G. Howard & J. S. Vitter, "Analysis of Arithmetic Coding for Data Compression," *Information Processing and Management* 28 (1992), 749–763.

[23] P. G. Howard & J. S. Vitter, "New Methods for Lossless Image Compression Using Arithmetic Coding," *Information Processing and Management* 28 (1992), 765–779.

[24] P. G. Howard & J. S. Vitter, "Design and Analysis of Fast Text Compression Based on Quasi-Arithmetic Coding," in *Proc. Data Compression Conference*, J. A. Storer & M. Cohn, eds., Snowbird, Utah, Mar. 30-Apr. 1, 1993, 98–107.

[25] D. A. Huffman, "A Method for the Construction of Minimum Redundancy Codes," *Proceedings of the Institute of Radio Engineers* 40 (1952), 1098–1101.

[26] D. E. Knuth, "Dynamic Huffman Coding," *J. Algorithms* 6 (June 1985), 163–180.

[27] G. G. Langdon, "Probabilistic and Q-Coder Algorithms for Binary Source Adaptation," in *Proc. Data Compression Conference*, J. A. Storer & J. H. Reif, eds., Snowbird, Utah, Apr. 8–11, 1991, 13–22.

[28] G. G. Langdon, "A Note on the Ziv-Lempel Model for Compressing Individual Sequences," *IEEE Trans. Inform. Theory* IT–29 (Mar. 1983), 284–287.

[29] G. G. Langdon & J. Rissanen, "Compression of Black-White Images with Arithmetic Coding," *IEEE Trans. Comm.* COM–29 (1981), 858–867.

[30] F. T. Leighton & R. L. Rivest, "Estimating a Probability Using Finite Memory," *IEEE Trans. Inform. Theory* IT–32 (Nov. 1986), 733–742.

[31] D. A. Lelewer & D. S. Hirschberg, "Streamlining Context Models for Data Compression," in *Proc. Data Compression Conference*, J. A. Storer & J. H. Reif, eds., Snowbird, Utah, Apr. 8–11, 1991, 313–322.

[32] V. S. Miller & M. N. Wegman, "Variations on a Theme by Ziv and Lempel," in *Combinatorial Algorithms on Words*, A. Apostolico & Z. Galil, eds., NATO ASI Series #F12, Springer-Verlag, Berlin, 1984, 131–140.

[33] J. L. Mitchell & W. B. Pennebaker, "Optimal Hardware and Software Arithmetic Coding Procedures for the Q-Coder," *IBM J. Res. Develop.* 32 (Nov. 1988), 727–736.

[34] A. M. Moffat, "Predictive Text Compression Based upon the Future Rather than the Past," *Australian Computer Science Communications* 9 (1987), 254–261.

[35] A. M. Moffat, "Word-Based Text Compression," *Software–Practice and Experience* 19 (Feb. 1989), 185–198.

[36] A. M. Moffat, "Linear Time Adaptive Arithmetic Coding," *IEEE Trans. Inform. Theory* IT–36 (Mar. 1990), 401–406.

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

[38] K. Mohiuddin, J. J. Rissanen & M. Wax, "Adaptive Model for Nonstationary Sources," *IBM Technical Disclosure Bulletin* 28 (Apr. 1986), 4798–4800.

[39] D. S. Parker, "Conditions for the Optimality of the Huffman Algorithm," *SIAM J. Comput.* 9 (Aug. 1980), 470–489.

[40] R. Pasco, "Source Coding Algorithms for Fast Data Compression," Stanford Univ., Ph.D. Thesis, 1976.

[41] W. B. Pennebaker & J. L. Mitchell, "Probability Estimation for the Q-Coder," *IBM J. Res. Develop.* 32 (Nov. 1988), 737–752.

[42] W. B. Pennebaker & J. L. Mitchell, "Software Implementations of the Q-Coder," *IBM J. Res. Develop.* 32 (Nov. 1988), 753–774.

[43] W. B. Pennebaker, J. L. Mitchell, G. G. Langdon & R. B. Arps, "An Overview of the Basic Principles of the Q-Coder Adaptive Binary Arithmetic Coder," *IBM J. Res. Develop.* 32 (Nov. 1988), 717–726.

[44] J. Rissanen, "Modeling by Shortest Data Description," *Automatica* 14 (1978), 465–571.

[45] J. Rissanen, "A Universal Prior for Integers and Estimation by Minimum Description Length," *Ann. Statist.* 11 (1983), 416–432.

[46] J. Rissanen, "Universal Coding, Information, Prediction, and Estimation," *IEEE Trans. Inform. Theory* IT–30 (July 1984), 629–636.

[47] J. Rissanen & G. G. Langdon, "Universal Modeling and Coding," *IEEE Trans. Inform. Theory* IT–27 (Jan. 1981), 12–23.

[48] J. J. Rissanen, "Generalized Kraft Inequality and Arithmetic Coding," *IBM J. Res. Develop.* 20 (May 1976), 198–203.

[49] J. J. Rissanen & G. G. Langdon, "Arithmetic Coding," *IBM J. Res. Develop.* 23 (Mar. 1979), 146–162.

[50] J. J. Rissanen & K. M. Mohiuddin, "A Multiplication-Free Multialphabet Arithmetic Code," *IEEE Trans. Comm.* 37 (Feb. 1989), 93–98.

[51] C. Rogers & C. D. Thomborson, "Enhancements to Ziv-Lempel Data Compression," Dept. of Computer Science, Univ. of Minnesota, Technical Report TR 89–2, Duluth, Minnesota, Jan. 1989.

[52] F. Rubin, "Arithmetic Stream Coding Using Fixed Precision Registers," *IEEE Trans. Inform. Theory* IT–25 (Nov. 1979), 672–675.

[53] B. Y. Ryabko, "Data Compression by Means of a Book Stack," *Problemy Peredachi Informatsii* 16 (1980).

[54] C. E. Shannon, "A Mathematical Theory of Communication," *Bell Syst. Tech. J.* 27 (July 1948), 398–403.

[55] J. S. Vitter, “Dynamic Huffman Coding,” *ACM Trans. Math. Software* 15 (June 1989), 158–167, also appears as Algorithm 673, Collected Algorithms of ACM, 1989.

[56] J. S. Vitter, “Design and Analysis of Dynamic Huffman Codes,” *Journal of the ACM* 34 (Oct. 1987), 825–845.

[57] 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.

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

[59] J. Ziv & A. Lempel, “A Universal Algorithm for Sequential Data Compression,” *IEEE Trans. Inform. Theory* IT–23 (May 1977), 337–343.

[60] J. Ziv & A. Lempel, “Compression of Individual Sequences via Variable Rate Coding,” *IEEE Trans. Inform. Theory* IT–24 (Sept. 1978), 530–536.