Arithmetic Coding Revisited

ALISTAIR MOFFAT
The University of Melbourne
RADFORD M. NEAL
University of Toronto
and
IAN H. WITTEN
The University of Waikato

Over the last decade, arithmetic coding has emerged as an important compression tool. It is now the method of choice for adaptive coding on multisymbol alphabets because of its speed, low storage requirements, and effectiveness of compression. This article describes a new implementation of arithmetic coding that incorporates several improvements over a widely used earlier version by Witten, Neal, and Cleary, which has become a *de facto* standard. These improvements include fewer multiplicative operations, greatly extended range of alphabet sizes and symbol probabilities, and the use of low-precision arithmetic, permitting implementation by fast shift/add operations. We also describe a modular structure that separates the coding, modeling, and probability estimation components of a compression system. To motivate the improved coder, we consider the needs of a word-based text compression program. We report a range of experimental results using this and other models. Complete source code is available.

Categories and Subject Descriptors: E.4 [**Data**]: Coding and Information Theory—*data compaction and compression*; E.1 [**Data**]: Data Structures

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Approximate coding, arithmetic coding, text compression, word-based model

This investigation was supported by the Australian Research Council and the Natural Sciences and Engineering Research Council of Canada. A preliminary presentation of this work was made at the 1995 IEEE Data Compression Conference.
Authors' addresses: A. Moffat, Department of Computer Science, The University of Melbourne, Parkville, Victoria 3052, Australia; email: alistair@cs.mu.oz.au; R. M. Neal, Department of Statistics and Department of Computer Science, University of Toronto, Canada; email: radford@cs.utoronto.ca; I. H. Witten, Department of Computer Science, The University of Waikato, Hamilton, New Zealand; email: ihw@waikato.ac.nz.

1. INTRODUCTION

During its long gestation in the 1970's and early 1980's, arithmetic coding [Rissanen 1976; Rissanen and Langdon 1979; Rubin 1979; Rissanen and Langdon 1981; Langdon 1984] was widely regarded more as a curiosity than a practical coding technique. This was particularly true for applications where the alphabet has many symbols, as Huffman coding is usually reasonably effective in such cases [Manstetten 1992]. One factor that helped arithmetic coding gain the popularity it enjoys today was the publication of source code for a multisymbol arithmetic coder by Witten et al. [1987] in *Communications of the ACM*, which we refer to as the CACM implementation. One decade later, our understanding of arithmetic coding has further matured, and it is timely to review the components of that implementation and summarize the improvements that have emerged. We also describe a novel, previously unpublished, method for performing the underlying calculation needed for arithmetic coding. Software is available that implements the revised method.

The major improvements discussed in this article and implemented in the software are as follows:

—Enhanced models that allow higher-performance compression.

—A more modular division into modeling, estimation, and coding subsystems.

—Data structures that support arithmetic coding on large alphabets.

—Changes to the coding procedure that reduce the number of multiplications and divisions and which permit most of them to be done with low-precision arithmetic.

—Support for larger alphabet sizes and for more accurate representations of probabilities.

—A reformulation of the decoding procedure that greatly simplifies the decoding loop and improves decoding speed.

—An extension providing efficient coding for binary alphabets.

To motivate these changes, we examine in detail the needs of a word-based model for text compression. While not the best-performing model for text (see, for example, the compression results listed by Witten et al. [1994]), word-based modeling combines several attributes that test the capabilities and limitations of an arithmetic coding system.

The new implementation of arithmetic coding is both more versatile and more efficient than the CACM implementation. When combined with the same character-based model as the CACM implementation, the changes that we advocate result in up to two-fold speed improvements, with only a small loss of compression. This faster coding will also be of benefit in any other compression system that makes use of arithmetic coding (such as the block-sorting method of Burrows and Wheeler [1994]), though the percent-

age of overall improvement will of course vary depending on the time used in other operations and on the exact nature of the hardware being used.

The new implementation is written in C, and is publicly available through the Internet by anonymous ftp, at `munnari.oz.au`, directory `/pub/arith_coder`, file `arith_coder.tar.Z` or `arith_coder.tar.gz`. The original CACM package [Witten et al. 1987] is at `ftp.cpsc.ucalgary.ca` in file `/pub/projects/ar.cod/cacm-87.shar`. Software that implements the new method for performing the arithmetic coding calculations, but is otherwise similar to the CACM version, can be found at `ftp.cs.toronto.edu` in the directory `/pub/radford`, file `lowp_ac.shar`.

In the remainder of this introduction we give a brief review of arithmetic coding, describe modeling in general, and word-based models in particular, and discuss the attributes that the arithmetic coder must embody if it is to be usefully coupled with a word-based model. Section 2 examines the interface between the model and the coder, and explains how it can be designed to maximize their independence. Section 3 shows how accurate probability estimates can be maintained efficiently in an adaptive compression system, and describes an elegant structure due to Fenwick [1994]. In Section 4 the CACM arithmetic coder is reexamined, and our improvements are described. Section 5 analyzes the cost in compression effectiveness of using low precision for arithmetic operations. Low-precision operations may be desirable because they permit a shift/add implementation, details of which are discussed in Section 6. Section 7 describes the restricted coder for binary alphabets, and examines a simple binary model for text compression. Finally, Section 8 reviews the results and examines the various situations in which arithmetic coding should and should not be used.

1.1 The Idea of Arithmetic Coding

We now give a brief overview of arithmetic coding. For additional background the reader is referred to the work of Langdon [1984], Witten et al. [1987; 1994], Bell et al. [1990], and Howard and Vitter [1992; 1994].

Suppose we have a message composed of symbols over some finite alphabet. Suppose also that we know the probability of appearance of each of the distinct symbols, and seek to represent the message using the smallest possible number of bits. The well-known algorithm of Huffman [1952] takes a set of probabilities and calculates, for each symbol, a code word that unambiguously represents that symbol. Huffman's method is known to give the best possible representation when all of the symbols must be assigned discrete code words, each an integral number of bits long. The latter constraint in effect means that all symbol probabilities are approximated by negative powers of two. In an arithmetic coder the exact symbol probabilities are preserved, and so compression effectiveness is better, sometimes markedly so. On the other hand, use of exact probabilities means that it is not possible to maintain a discrete code word for each symbol; instead an overall code for the whole message must be calculated.

The mechanism that achieves this operates as follows. Suppose that p_i is the probability of the ith symbol in the alphabet, and that variables L and R are initialized to 0 and 1 respectively. Value L represents the smallest binary value consistent with a code representing the symbols processed so far, and R represents the product of the probabilities of those symbols. To encode the next symbol, which (say) is the jth of the alphabet, both L and R must be refined: L is replaced by $L + R\Sigma_{i=1}^{j-1} p_i$ and R is replaced by $R \cdot p_j$, preserving the relationship between L, R, and the symbols so far processed. At the end of the message, any binary value between L and $L + R$ will unambiguously specify the input message. We transmit the shortest such binary string, c. Because c must have at least $-\lceil \log_2 R \rceil$ and at most $-\lceil \log_2 R \rceil + 2$ bits of precision, the procedure is such that a symbol with probability p_j is effectively coded in approximately $-\log_2 p_j$ bits, thereby meeting the entropy-based lower bound of Shannon [1948].

This simple description has ignored a number of important problems. Specifically, the process described above requires extremely high precision arithmetic, since L and R must potentially be maintained to a million bits or more of precision. We may also wonder how best to calculate the cumulative probability distribution, and how best to perform the arithmetic. Solving these problems has been a major focus of past research, and of the work reported here.

1.2 The Role of the Model

The CACM implementation [Witten et al. 1987] included two driver programs that coupled the coder with a static zero-order character-based model, and with a corresponding adaptive model. These were supplied solely to complete a compression program, and were certainly not intended to represent excellent models for compression. Nevertheless, several people typed in the code from the printed page and compiled and executed it, only—much to our chagrin—to express disappointment that the new method was inferior to widely available benchmarks such as *Compress* [Hamaker 1988; Witten et al. 1988].

In fact, all that the CACM article professed to supply was a state-of-the art coder with two simple, illustrative, but mediocre models. One can think of the model as the "intelligence" of a compression scheme, which is responsible for deducing or interpolating the structure of the input, whereas the coder is the "engine room" of the compression system, which converts a probability distribution and a single symbol drawn from that distribution into a code [Bell et al. 1990; Rissanen and Langdon 1981]. In particular, the arithmetic coding "engine" is independent of any particular model. The example models in this article are meant purely to illustrate the demands placed upon the coder, and to allow different coders to be compared in a uniform test harness. Any improvements to the coder will

primarily yield better compression *efficiency*, that is, a reduction in time or space usage. Improvements to the model will yield improved compression *effectiveness*, that is, a decrease in the size of the encoded data. In this article we are primarily interested in compression efficiency, although we will also show that the approximations inherent in the revised coder do not result in any substantial loss of compression effectiveness.

The revised implementation does, however, include a more effective word-based model [Bentley et al. 1986; Horspool and Cormack 1992; Moffat 1989], which represents the stream as a sequence of words and nonwords rather than characters, with facilities for spelling out new words as they are encountered using a subsidiary character mode. Since the entropy of words in typical English text is around 10–15 bits each, and that of nonwords is around 2–3 bits, between 12 and 18 bits are required to encode a typical five-character word and the following one-character nonword. Large texts are therefore compressed to around 30% of their input size (2.4 bits per character)—a significant improvement over the 55%–60% (4.4–4.8 bits per character) achieved by zero-order character-based models of English. Witten et al. [1994] give results comparing character-based models with word-based models.

A word-based compressor can also be faster than a character-based one. Once a good vocabulary has been established, most words are coded as single symbols rather than as character sequences, reducing the number of time-consuming coding operations required.

What is more relevant, for the purposes of this article, is that word-based models illustrate several issues that do not arise with character-based models:

—An efficient data structure is needed to accumulate frequency counts for a large alphabet.

—Multiple coding contexts are necessary, for tokens, characters, and lengths, for both words and nonwords. Here, a coding context is a conditioning class on which the probability distribution for the next symbol is based.

—An escape mechanism is required to switch from one coding context to another.

—Data structures must be resizable because there is no a priori bound on alphabet size.

All of these issues are addressed in this article.

Arithmetic coding is most useful for adaptive compression, especially with large alphabets. This is the application envisioned in this article, and in the design of the new implementation. For static and semistatic coding, in which the probabilities used for encoding are fixed, Huffman coding is usually preferable to arithmetic coding [Bookstein and Klein 1993; Moffat and Turpin 1997; Moffat et al. 1994].

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.