Arithmetic Coding for Data Compression

Article *in* Proceedings of the IEEE · July 1994

DOI: 10.1109/5.286189 · Source: IEEE Xplore

Jeffrey Scott Vitter
University of Mississippi

CS--1994--09

Arithmetic Coding for Data Compression

Paul G. Howard, Jeffrey S. Vitter

Department of Computer Science

Duke University

Durham, North Carolina 27708-0129

March 25, 1994

Arithmetic Coding for Data Compression

PAUL G. HOWARD AND JEFFREY SCOTT VITTER, FELLOW, IEEE

Arithmetic coding provides an effective mechanism for removing redundancy in the encoding of data. We show how arithmetic coding works and describe an efficient implementation that uses table lookup as a fast alternative to arithmetic operations. The reduced-precision arithmetic has a provably negligible effect on the amount of compression achieved. We can speed up the implementation further by use of parallel processing. We discuss the role of probability models and how they provide probability information to the arithmetic coder. We conclude with perspectives on the comparative advantages and disadvantages of arithmetic coding.

Index terms— Data compression, arithmetic coding, lossless compression, text modeling, image compression, text compression, adaptive, semi-adaptive.

I. Arithmetic coding

The fundamental problem of lossless compression is to decompose a data set (for example, a text file or an image) into a sequence of events, then to encode the events using as few bits as possible. The idea is to assign short codewords to more probable events and longer codewords to less probable events. Data can be compressed whenever some events are more likely than others. Statistical coding techniques use estimates of the probabilities of the events to assign the codewords. Given a set of mutually distinct events e_1, e_2, e_3, …, e_n, and an accurate assessment of the probability distribution P of the events, Shannon [1] proved that the the smallest possible expected number of bits needed to encode an event is the *entropy* of P, denoted by

$$H(P) = \sum_{k=1}^{n} -p\{e_k\} \log_2 p\{e_k\},$$

where $p\{e_k\}$ is the probability that event e_k occurs. An optimal code outputs $-\log_2 p$ bits to encode an event whose probability of occurrence is p. Pure arithmetic codes supplied with accurate probabilities provides optimal compression. The older and better-known Huffman codes [2] are optimal only among instantaneous codes, that is, those in which the encoding of one event can be decoded before encoding has begun for the next event.

In theory, arithmetic codes assign one "codeword" to each possible data set. The codewords consist of half-open subintervals of the half-open unit interval $[0, 1)$, and are expressed by specifying enough bits to distinguish the subinterval corresponding to the actual data set from all other possible subintervals. Shorter codes correspond to larger subintervals and thus more probable input data sets. In practice, the subinterval is refined incrementally using the probabilities of the individual events, with bits being output as soon as they are known. Arithmetic codes almost always give better compression than prefix codes, but they lack the direct correspondence between the events in the input data set and bits or groups of bits in the coded output file.

A statistical coder must work in conjunction with a modeler that estimates the probability of each possible event at each point in the coding. The probability model need not describe the process that generates the data; it merely has to provide a probability distribution for the data items. The probabilities do not even have to be particularly accurate, but the more accurate they are, the better the compression will be. If the probabilities are wildly inaccurate, the file may even be expanded rather than compressed, but the original data can still be recovered. To obtain maximum compression of a file, we need both a good probability model and an efficient way of representing (or learning) the probability model.

To ensure decodability, the encoder is limited to the use of model information that is available to the decoder. There are no other restrictions on the model; in particular, it can change as the file is being encoded. The models can be *adaptive* (dynamically estimating the probability of each event based on all events that precede it), *semi-adaptive* (using a preliminary pass of the input file to gather statistics), or *non-adaptive* (using fixed probabilities for all files). Non-adaptive models can perform arbitrarily poorly [3]. Adaptive codes allow one-pass coding but require a more complicated data structure. Semi-adaptive codes require two passes and transmission of model data as side information; if the model data is transmitted efficiently they can provide slightly better compression than adaptive codes, but in general the cost of transmitting the model is about the same as the "learning" cost in the adaptive case [4].

To get good compression we need models that go beyond global event counts and take into account the structure of the data. For images this usually means using the numeric intensity values of nearby pixels to predict the intensity of each new pixel and using a suitable probability distribution for the residual error to allow for noise and variation between regions within the image. For text, the previous letters form a context, in the manner of a Markov process.

In Section II, we provide a detailed description of pure arithmetic coding, along with an example to illustrate the process. We also show enhancements that allow incremental transmission and fixed-precision arithmetic. In Section III we extend the fixed-precision idea to low precision, and show how we can speed up arithmetic coding with little degradation of compression performance by doing all the arithmetic ahead of time and storing the results in lookup tables. We call the resulting procedure *quasi-arithmetic coding*. In Section IV we briefly explore the possibility of parallel coding using quasi-arithmetic coding. In Section V we discuss the modeling process, separating it into structural and probability estimation components. Each component can be adaptive, semi-adaptive, or static; there are two approaches to

Manuscript received Mmm DD, 1993. Some of this work was performed while both authors were at Brown University and while the first author was at Duke University. Support was provided in part by NASA Graduate Student Researchers Program grant NGT–50420, 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 Universities Space Research Association/CESDIS associate memberships.

P. G. Howard is with AT&T Bell Laboratories, Visual Communications Research, Room 4C–516, 101 Crawfords Corner Road, Holmdel, NJ 07733–3030.

J. S. Vitter is with the Department of Computer Science, Duke University, Box 90129, Durham, NC 27708–0129.

vides a discussion of the advantages and disadvantages of arithmetic coding and suggestions of alternative methods.

II. How arithmetic coding works

In this section we explain how arithmetic coding works and give operational details; our treatment is based on that of Witten, Neal, and Cleary [5]. Our focus is on encoding, but the decoding process is similar.

A. Basic algorithm for arithmetic coding

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

1. We begin with a "current interval" $[L, H)$ initialized to $[0, 1)$.
2. For each event in the file, we perform two steps.
   (a) We subdivide the current interval into subintervals, one for each possible event. The size of a event's subinterval is proportional to the estimated probability that the event will be the next event in the file, according to the model of the input.
   (b) We select the subinterval corresponding to the event that actually occurs next, 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 events, which is the probability p of the particular sequence of events in the file. The final step uses at most $\lfloor -\log_2 p \rfloor + 2$ 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 event coded just once, or some external indication of the file's length. Either method adds only a small amount to the code length.

In step 2, we need to compute only the subinterval corresponding to the event a_i that actually occurs. To do this it is convenient to use two "cumulative" probabilities: the cumulative probability $P_C = \sum_{k=1}^{i-1} p_k$ and the next-cumulative probability $P_N = P_C + p_i = \sum_{k=1}^{i} p_k$. 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, especially when many more than two events are possible.

We now provide an example, repeated a number of times to illustrate different steps in the development of arithmetic coding. For simplicity we choose between just two events at each step, although the principles apply to the multi-event case as well. We assume that we know *a priori* that we have a file consisting of three events (or three letters in the case of text compression); the first event is either a_1 (with probability $p\{a_1\} = \frac{2}{3}$) or b_1 (with probability $p\{b_1\} = \frac{1}{3}$); the second event is a_2 (with probability $p\{a_2\} = \frac{1}{2}$) or b_2 (with probability $p\{b_2\} = \frac{1}{2}$); and the third event is a_3 (with probability $p\{a_3\} = \frac{3}{5}$) or b_3 (with probability $p\{b_3\} = \frac{2}{5}$). The actual file to be encoded is the sequence $b_1 a_2 b_3$.

The steps involved in pure arithmetic coding are illustrated in Table 1 and Fig. 1. In this example the final interval corresponding to the actual file $b_1 a_2 b_3$ is $[\frac{23}{30}, \frac{5}{6})$. The length of the interval is $\frac{1}{15}$, which is the probability of $b_1 a_2 b_3$, computed by multiplying the probabilities of the three events: $p\{b_1\}p\{a_2\}p\{b_3\} = \frac{1}{3}\frac{1}{2}\frac{2}{5} = \frac{1}{15}$. In binary, the final interval is $[0.110001\ldots, 0.110101\ldots)$. Since all binary numbers that begin with 0.11001 are entirely within this interval, out-

Table 1 Example of pure arithmetic coding

Action	Subintervals
Start	$[0, 1)$
Subdivide with left prob. $p\{a_1\} = \frac{2}{3}$	$[0, \frac{2}{3}), [\frac{2}{3}, 1)$
Input b_1, select right subinterval	$[\frac{2}{3}, 1)$
Subdivide with left prob. $p\{a_2\} = \frac{1}{2}$	$[\frac{2}{3}, \frac{5}{6}), [\frac{5}{6}, 1)$
Input a_2, select left subinterval	$[\frac{2}{3}, \frac{5}{6})$
Subdivide with left prob. $p\{a_3\} = \frac{3}{5}$	$[\frac{2}{3}, \frac{23}{30}), [\frac{23}{30}, \frac{5}{6})$
Input b_3, select right subinterval	$[\frac{23}{30}, \frac{5}{6})$
	$= [0.110001\ldots_2, 0.110101\ldots_2)$
Output **11001**	0.11001_2 is the shortest binary fraction that lies within $[\frac{23}{30}, \frac{5}{6})$

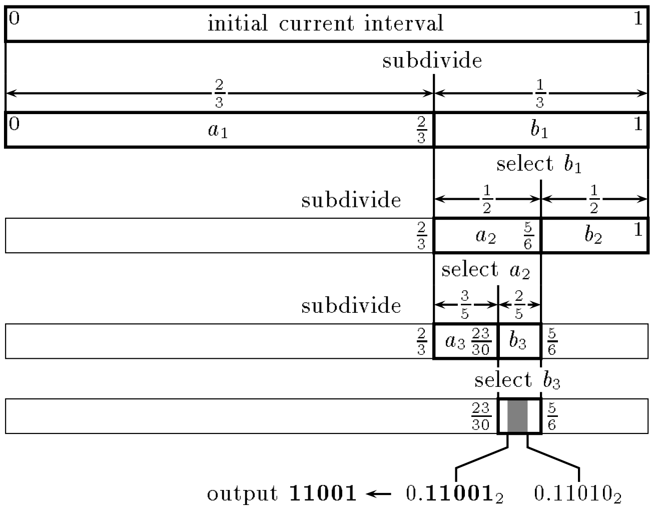

Fig. 1. Pure arithmetic coding graphically illustrated

B. Incremental output

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 [5] add a clever mechanism for preventing the current interval from shrinking too much when the endpoints are close to $\frac{1}{2}$ but straddle $\frac{1}{2}$. 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 $\frac{1}{2}$. This follow-on procedure may be repeated any number of times, so the current interval size is always strictly longer than $\frac{1}{4}$.

Mechanisms for incremental transmission and fixed precision arithmetic have been developed through the years by Pasco [6], Rissanen [7], Rubin [8], Rissanen and Langdon [9], Guazzo [10], and Witten, Neal, and Cleary [5]. The bit-stuffing idea of Langdon and others at IBM that limits the propagation of carries in the additions serves a function

Table 2 Example of pure arithmetic coding with incremental transmission and interval expansion

Action	Subintervals
Start	$[0, 1)$
Subdivide with left prob. $p\{a_1\} = \frac{2}{3}$	$[0, \frac{2}{3}), [\frac{2}{3}, 1)$
Input b_1, select right subinterval	$[\frac{2}{3}, 1)$
Output **1**, expand	$[\frac{1}{3}, 1)$
Subdivide with left prob. $p\{a_2\} = \frac{1}{2}$	$[\frac{1}{3}, \frac{2}{3}), [\frac{2}{3}, 1)$
Input a_2, select left subinterval	$[\frac{1}{3}, \frac{2}{3})$
Increment **follow** count, expand	$[\frac{1}{6}, \frac{5}{6})$
Subdivide with left prob. $p\{a_3\} = \frac{3}{5}$	$[\frac{1}{6}, \frac{17}{30}), [\frac{17}{30}, \frac{5}{6})$
Input b_3, select right subinterval	$[\frac{17}{30}, \frac{5}{6})$
Output **1**, output **0** (**follow** bit), expand	$[\frac{2}{15}, \frac{2}{3})$
Output **01**	$[\frac{1}{4}, \frac{1}{2})$ is entirely within $[\frac{2}{15}, \frac{2}{3})$

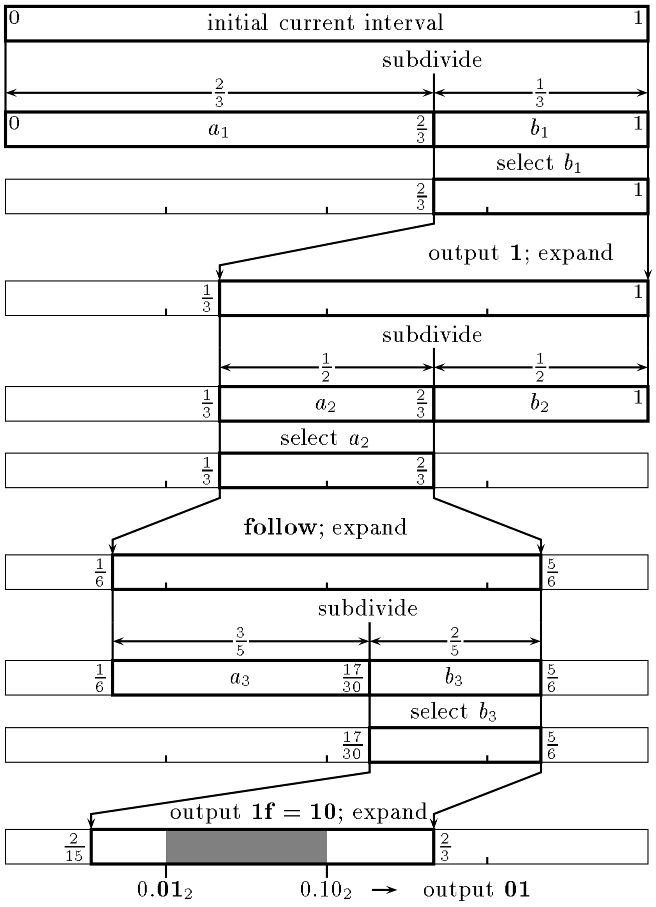

Fig. 2. Pure arithmetic coding with incremental transmission and interval expansion, graphically illustrated

We now describe in detail how the incremental output and interval expansion work. We add the following step immediately after the selection of the subinterval corresponding to an input event, step 2(b) in the basic algorithm above.

2. (c) We repeatedly execute the following steps in sequence until the loop is explicitly halted:
   1. If the new subinterval is not entirely within one of the intervals $[0, \frac{1}{2})$, $[\frac{1}{4}, \frac{3}{4})$, or $[\frac{1}{2}, 1)$, we exit the loop and return.
   2. If the new subinterval lies entirely within $[0, \frac{1}{2})$, we output **0** and any following **1**s left over from previous events; then we double the size of the subinterval by linearly expanding $[0, \frac{1}{2})$ to $[0, 1)$.
   3. If the new subinterval lies entirely within $[\frac{1}{2}, 1)$, we output **1** and any following **0**s left over from previous events; then we double the size of the subinterval by linearly expanding $[\frac{1}{2}, 1)$ to $[0, 1)$.
   4. If the new subinterval lies entirely within $[\frac{1}{4}, \frac{3}{4})$, we keep track of this fact for future output by incrementing the **follow** count; then we double the size of the subinterval by linearly expanding $[\frac{1}{4}, \frac{3}{4})$ to $[0, 1)$.

Table 2 and Fig. 2 illustrate this process. In the example, interval expansion occurs exactly once for each input event, but in other cases it may occur more than once or not at all. The follow-on procedure is applied when processing the second input event a_2. The **1** output after processing the third event b_3 is therefore followed by its complement **0**. The final interval is $[\frac{2}{15}, \frac{2}{3})$. Since all binary numbers that start with 0.01 are within this range, outputting **01** suffices to uniquely identify the range. The encoded file is **11001**, as before. This is no coincidence: the computations are essentially the same. The final interval is eight times as long as in the previous example because of the three doublings of the current interval.

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 $-\log_2 a$ bits not yet output. In the basic method the state contains *all* the information about the output, since nothing is output until the end. In the incremental implementation, the state always contains fewer than two bits of output information, since the length of the current interval is always more than $\frac{1}{4}$. The final state in the incremental example is $[\frac{2}{15}, \frac{2}{3})$, which contains $-\log_2 \frac{8}{15} \approx 0.907$ bits of information; the final two output bits are needed to unambiguously transmit this information.

C. Use of integer arithmetic

In practice, the arithmetic can be done by storing the endpoints of the current interval as sufficiently large integers rather than in floating point or exact rational numbers. We also use integers for the frequency counts used to estimate event probabilities. The subdivision process involves selecting non-overlapping intervals (of length at least 1) with lengths approximately proportional to the counts. Table 3 illustrates the use of integer arithmetic using a full interval of $[0, N) = [0, 1024)$. (The graphical version of Table 3 is essentially the same as Fig. 2 and is not included.) The length of the current interval is always at least $N/4 + 2$, 258 in this case, so we can always use probabilities precise to at least 1/258; often the precision will be near 1/1024. In practice we use even larger integers; the interval $[0, 65\,536)$ is a common choice, and gives a practically continuous choice of probabilities at each step. The subdivisions in this example are not quite the same as those in Table 2 because the resulting intervals are rounded to integers. The encoded file is **11001** as before, but for a longer input file the encodings

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.

WHAT WILL YOU BUILD? | **sales@docketalarm.com** | **1-866-77-FASTCASE**

fastcase® Smarter legal research.