A Multiplication-Free Multialphabet Arithmetic Code

JORMA RISSANEN AND K. M. MOHIUDDIN

Abstract—A new recursion for arithmetic codes used for data compression is described which requires no multiplication or division, even in the case with nonbinary alphabets. For this reason, such a code admits a simple and fast hardware implementation. The inputs to the code are, in addition to the symbols to be encoded, either the symbol probabilities or, more simply, the corresponding occurrence counts. Hence, the code is applicable in conjunction with stationary and nonstationary models alike. The code efficiency is typically in the range of 97–99 percent.

I. Introduction

ARITHMETIC codes, introduced in the LIFO-form (last-in first-out) in Rissanen [3] and subsequently modified to the important FIFO-form (first-in first-out) in Pasco [2], are by far the most flexible and powerful coding techniques available for the purpose of achieving compression, for they can be used with equal ease to encode strings modeled by stationary or nonstationary sources. The basic encoding operation in an arithmetic code requires an update of the probability $P(s)$ of the so-far processed string s, which can be done by multiplication $P(s)P(i|s)$ where $P(i|s)$ is a conditional probability of the symbol i given s. However, the multiplication in a hardware implementation of the code is relatively expensive and slow even though both probabilities are numbers which in their binary representation have at most a fixed number of significant digits. In the special case with a binary alphabet, the multiplication was replaced in Langdon and Rissanen [1] by the much simpler shift operation, which was made possible by an approximation of the smaller symbol probability with an integral power of 1/2. However, the same idea does not generalize to nonbinary alphabets for the reason that the symbol probabilities cannot be approximated well enough as powers of 1/2.

In this paper, we describe a new implementation of arithmetic codes in which the proper multiplication is avoided even with nonbinary alphabets. As an added bonus the code can accept as inputs the occurrence counts of the symbols in place of probabilities. This means that no division is needed to convert the counts into probabilities. Because of the reduced complexity, this algorithm is well suited for implementation in VLSI.

We give an analysis of the code efficiency and discuss the choice of various parameters. We have also studied the code efficiency experimentally. In typical strings with the alphabet sizes varying from 2 to 84, the efficiency was found to be 97–99 percent or better. A particularly simple implementation results in the case with a binary alphabet for which the efficiency of our code exceeds that of the fast code in Langdon and Rissanen [1] mentioned above.

II. Review of Arithmetic Codes

Let the alphabet consist of the symbols $i = 0, 1, \cdots, m$. An arithmetic code encodes the string to be compressed, symbol for symbol from left to right. When encoding the symbol i, immediately following the so-far processed string s, the code requires as input the data that represent the conditional probability of the symbol's occurrence at its context, which in the most general case is the entire past string s. Such parameters are provided by the so-called modeling unit. In the special case where the string is modeled by a stationary source, these parameters do not depend on s. Often, the simplest way to represent the model parameters is in terms of integer-valued symbol occurrence counts, say n_i for the symbol i in its context, which we from here on do not indicate. These numbers need not correspond to the real occurrences of the symbols; instead, the modeling unit may update them in a sophisticated manner to reflect changing statistics. If n denotes the sum of the counts n_i, then we may say that the symbol i following string s is modeled by the probability $P(i|s) = n_i/n$. In addition to these counts, the modeling unit collects and delivers also the cumulative sums $Q(i) = n_0 + \cdots + n_{i-1}$, $i = 1, \cdots, m$, $Q(0) = 0$. The modeling unit should update for each symbol i not only the count n_i but also the cumulative counts $Q(j)$, $j > i$, which are affected by n_i. Finally, because of the approximations that we introduce later, we assume for the best compression efficiency that the symbol with the highest count to be the last, i.e.,

$$n_m \geq n_i, \text{ all } i. \tag{2.1}$$

An arithmetic code constructs the code string as a cumulative probability of the strings that precede the considered one in the lexical order of the strings. When calculating this cumulative probability, the code uses a certain approximation of the probability $P(s)$ defined by the model, which we denote by $A(s)$. This approximation satisfies all the properties of an information source, as explained below. The lexical order on the strings implies, in particular, that if $s' < s$, both strings have the same length, then for all symbols i, $s'i < s0$ where si denotes the string consisting of s followed by the symbol i. Therefore, if $C(s)$ denotes the binary fractional number representing the code, i.e., the cumulative probability of the string s, then the code of the one symbol longer string si is given by

$$C(s0) = C(s)$$
$$C(si) = C(s) + A(s0) + \cdots + A(si-1), \; i > 0 \tag{2.2}$$

where for the sake of clarity the notation $si - 1$ was used for the string consisting of s followed by the symbol $i - 1$. The initial conditions are $C(\lambda) = 0$ and $A(\lambda) = 1$ where λ denotes the empty string. The truth of this recursion for the binary alphabet becomes immediate if the reader cares to draw a binary tree with the root up, and at each internal node 0 points to the left son and 1 to the right son. Then the ordering of the strings of length n grows from left to right, and the formation of the cumulative string probabilities indeed follows the given recursion. We shall also see to it that

$$A(s) = A(s0) + \cdots + A(sm) \tag{2.3}$$

which ensures decodability (Rissanen and Langdon [4]) which

Paper approved by the Editor for Coding Theory and Applications of the IEEE Communications Society. Manuscript received April 4, 1986; revised August 24, 1987.

The authors are with IBM Almaden Research Center, San Jose, CA 95120.

IEEE Log Number 8825337.

0090-6778/89/0200-0093$01.00

with its initialized value 1 causes $A(s)$ to define a bona fide information source. These equations leave open the issue of how $A(s)$ is computed recursively, which will be dealt with later.

The arithmetic operations called for by the two recursions will be done in two fixed size registers with width w (typically, $w = 16$ or 12), one for $C(s)$ and the other for $A(s)$. This is possible if we introduce the normalized quantities $a(s) = 2^{L(s)} A(s)$ where $L(s)$ is a nonnegative integer determined by the requirement that $a(s)$ is a w digit binary number in a fixed range. The selection of the range is discussed under Section V. For ease of implementation, the following range turns out to be suitable:

$$0.75 \le a(s) < 1.5. \tag{2.4}$$

This normalization is easy to do by means of shift operations. To do the encoding/decoding in a single cycle, special hardware should be provided to do shifts by multiple bit-positions in a single cycle. In these notations, the recursion (2.3) reads

$$a(s) = \bar{a}(s0) + \cdots + \bar{a}(sm) \tag{2.5}$$

where $\bar{a}(si) = 2^{L(s)} A(si)$. As s increases in length, $A(s)$ decreases as by (2.3) and $L(s)$ increases, and the quantities added to the code string according to (2.2) move to the right. Equivalently, the code string $C(s)$is shifted left relative to the fixed register where the addition takes place.

III. Binary Code

A particularly simple implementation results in the case of binary alphabets, which we describe first. Let x denote the symbol to which the model assigns the lower count, i.e., $n_x \le n/2$. This *low probability* symbol, of course, depends on the past string. Denote by x' the opposite *high probability* symbol. First, convert the integral count n into a fractional number in the range of (2.4). This is achieved by shifting n by enough positions $k(n)$ to the left such that the resultant number $\bar{n}$ will be in the range (2.4). Let the counts be maintained in registers of width w. Let $l(n)$ by the number of significant binary digits in n. If the second significant digit of n is 1 (the first of course being 1 by definition) then $k(n)$, the number of positions of shift, is equal to $(w - 1) - l(n)$. If, however, the significant digits of n start as $10\cdots$, then $k(n)$ is equal to $w - l(n)$. In the former case, one can think of the radix point as being just to the left of the most significant digit. Correspondingly, in the latter case, the radix point is just to the right of the most significant digit. For example, assume that the width $w = 8$. If $n = 00001101$ (binary), then $\bar{n} = 0.1101000$, but if $n = 00001011$ then $\bar{n} = 1.0110000$. In both cases, $l(n) = 4$. In the first case, n is shifted by $(8 - 1) - 4 = 3$ positions and in the second case by $8 - 4 = 4$ positions. Notice that for gaining the maximum speed in a VLSI implementation, these shifts have to be done in a single cycle using special hardware such as barrel shifters. We will define $\bar{n} = 2^{-k(n)} n$. Further define $\bar{n}_l = 2^{-k(n)} n_l$. Notice that the individual count n_x is shifted by the same number of positions $k(n)$ as n. In such cases where the inputs are available not as counts but as probabilities, each written with $w - 1$ fractional digits, we put $n = \bar{n} = 1$, which gives $\bar{n}_x = p_x$. Since $n = n_x + n_{x'}$, we have

$$\bar{n} = \bar{n}_x + \bar{n}_{x'}. \tag{3.1}$$

A recursion that would satisfy the basic decomposition of (2.5) (for $m = 1$) is

$$a(sx) = a(s)p(x|s), \quad a(sx') = a(s) - a(s)p(x|s) \tag{3.2}$$

where $p(x|s)$ is the conditional probability of x given the past string s. The goal is to apportion $a(s)$ into $a(si)$ in the proportion of the conditional probabilities of i given s. However, it is exactly the above multiplication that we would like to avoid. We first approximate the ideal conditional probability by $p(x|s) = n_x/n = \bar{n}_x/\bar{n}$ and then rewrite the recursion for $a(sx)$ from (3.2) as

$$a(sx) = a(s)\frac{\bar{n}_x}{\bar{n}} = \bar{n}_x \frac{a(s)}{\bar{n}}. \tag{3.3}$$

Since $a(s)$ and $\bar{n}$ are in the same range of (2.4), their ratio is of the order of unity. We can now approximate the recursion for $a(sx)$ and $a(sx')$ as

$$a(sx) = \bar{n}_x, \; a(sx') = a(s) - \bar{n}_x \tag{3.4}$$

and then normalize the new value of $a(s)$ to be in the range of (2.4), quite analogously with the general normalization with which we bring the addends $A(s)$ to the same range. In case the inputs are available as probabilities, our approximation is equivalent to the following recursions:

$$a(sx) = p(x|s), \quad a(sx') = 1 - p(x|s). \tag{3.5}$$

Let us illustrate the approximations with an example. Suppose $n = 1101$ and $n_0 = 0010$ in binary representation. By normalization of the counts $\bar{n} = 0.1101$ and $\bar{n}_0 = 0.0010$. Let, for example, $a(s) = 1.001$. Using the recursion of (3.4), $\bar{a}(s0) = 0.0010$ and $\bar{a}(s1) = 1.000$. Even though 0.001/1.001 differs from the count ration 0010/1101 by about 50 percent, the effect to the code length turns out to be small. The issue of the efficiency of the code is dealt with in greater detail in Section V.

We now describe the coding operations in terms of the actions in two registers, C and A, both having width w. We interpret the content of C as being a fractional binary number resulting when the binary point is placed to its left end, while in register A we place the binary point after the first position. The code string $C(s)$ consists of the symbols that get emitted from the left end of register C together with the symbols in C.

A. Encoding Algorithm

Initially fill register C with 0's and set A to $10\cdots0$.

1) Read the next symbol. If none exists, shift the contents of C left w positions to obtain the final string.

2) If the next symbol is the low probability symbol x, replace the contents of A by $\bar{n}_x$ and go to 4).

3) If the next symbol is the high probability symbol, add to register C the number $\bar{n}_x$, and subtract from register A the same number. Go to 4).

4) Shift the contents of both registers A and C left as many positions as required to bring A to the range [0.75, 1.5). Go to 1).

As numbers get added to register C, a carry-over may occur. A special *bit stuffing* routine may be used to handle such carries (Langdon and Rissanen [1]). Also, in order to reduce the frequency of the carry occurrences a *guard register* of width w_c may be placed to the left end of C to take care of carries shorter than w_c. In such a case, a corresponding register should be used at the decoder also.

We illustrate the encoding algorithm using a simple example. Let $n = 1000$, $n_0 = 0010$, and $n_1 = 0110$. After normalization, we have $\bar{n} = 1.000$, $\bar{n}_0 = 0.010$, and $\bar{n}_1 = 0.110$. We assume the register width w to be 4 bits. Let the string to be encoded be 0111010. Table I lists the contents of both C and A registers after each symbol has been encoded.

The decoding is done by virtually reversing the steps in the encoding.

B. Decoding Algorithm

Initialize C with w leftmost symbols of the code string and A with $10\cdots0$.

TABLE I
ENCODING EXAMPLE FOR BINARY ALPHABET

Symbol	*Reg C*	*Reg A*
Start	0000	1000
0	00 0000	1000
1	00 0100	0110
1	001 0000	1000
1	001 0100	0110
0	00101 0000	1000
1	00101 0100	0110
0	0010101 0000	1000

TABLE II
DECODING EXAMPLE FOR BINARY ALPHABET

Decoded Symbol	*Reg C*	*Reg A*
Start	0010	1000
0	1010	1000
1	0110	0110
1	0101	1000
1	0001	0110
0	0100	1000
1	0000	0110
0	0000	1000

1) Form an auxiliary quantity T by subtracting $\bar{n}_x$ from the contents of C. Test if $T < 0$.

2) If $T < 0$, decode the symbol as x. Load A with $\bar{n}_x$. Go to 4).

3) If $T \geq 0$, decode the symbol as the high probability symbol. Load C with T, subtract $\bar{n}_x$ from A, and go to 4).

4) Shift both registers left by as many positions as required to bring A to the range [0.75, 1.5), and read the same number of symbols from the code string in the vacated positions in C. If none exists, stop. Else, go to 1).

Let us use the coding parameters of the encoding-example and decode the string generated by the encoder. Table II lists the contents of both registers after each symbol has been decoded.

Notice in Step 2) of the encoding algorithm that the processing of the low probability symbol requires only one operation while the encoding of the high probability symbol in Step 3) requires two. However, these two operations can be done in parallel, which means that the processing of each symbol in hardware implementation requires about the same time. In a software implementation, which does not allow parallel processing, we get a faster encoding algorithm if we rearrange the coding operations as follows:

$$C(sx') = C(s), \quad C(sx) = C(s) + 2^{-L(sx')}(a(s) - \bar{n}_x). \quad (3.3)$$

In other words, the two serial operations, the subtraction in register A and then the addition in register C, are done for the low probability symbol, while for the more frequently occurring high probability symbol only the updating of the A register needs to be done. This means, though, that proper decoding is possible only when a dummy low probability symbol is added to the very end of the source string.

IV. Nonbinary Code

The key idea in the updates of the normalized addends $a(s)$ in the binary case was to scale the total symbol count so as to bring it close to $a(s)$, which then allows us to define the next addends simply as $\bar{n}_x$ and $a(s) - \bar{n}_x$, respectively. The same idea can be applied in the general case as well. We recall that the modeling unit places the symbol with the highest count as the last symbol m, which is done for the reason to reduce the approximation errors; the ordering of the other symbols does not matter. As before, we define $k(n)$ as the value of the integer k which brings $2^{-k}n$ in the range [0.75, 1.5). Then, we define the normalized counts

$$\bar{n} = 2^{-k(n)}n$$

$$\bar{n}_i = 2^{-k(n)}n_i$$

$$\bar{Q}(i) = \sum_{j<i} \bar{n}_j, \quad \bar{Q}(0) = 0.$$

In case the inputs are probabilities, each written with $w - 1$ fractional binary digits, we put $n = \bar{n} = 1$ and $\bar{n}_i = p_i$.

The idea for updating the addends is to define them as $a(si) = \bar{n}_i$, $i < m$, and $a(sm) = a(s) - \bar{Q}(m)$. But because we may have $\bar{n} > a(s)$, the last addend $a(sm) = a(s) - \bar{Q}(m)$ may be negative. In that case, we replace $k(n)$ by $k(n) + 1$, and this guarantees positivity of all the addends. The final encoding algorithm receives $\bar{n}_i$ and $\bar{Q}_i$ as the inputs, and then progresses by the following algorithm using two registers C and A. The content of C register is interpreted as a binary fractional number resulting when the binary point is placed to the left end, while in A the binary point is placed right after the first position. Hence, for example, if register C has width $w = 4$ then its content 0100 represents the number 0.0100, while in A the same content represents the number 0.100.

A. Encoding Algorithm

Initially fill register C with 0's and set A to $10 \cdots 0$.

1) Read the next symbol i. If none exists, shift the contents of C left w positions and stop. Else, test if $\bar{Q}(m) < A$. If true, put $\beta = 0$. Otherwise, put $\beta = 1$.

2) If the symbol i is not the last m add the number $2^{-\beta}\bar{Q}(i)$ to the contents of register C, load register A with the number $2^{-\beta}\bar{n}_i$, and normalize, Step 4).

3) If the symbol in the last, i.e., $i = m$, add the number $2^{-\beta}\bar{Q}(m)$ to the contents of register C, subtract the same number from A, and normalize, Step 4).

4) Shift the contents in both registers C and A left as many positions as needed to bring the contents of A to the range [0.75, 1.5), and fill the vacated positions with 0. Go to 1).

Whenever register C is shifted left, the bits "falling" off the left end are the successive symbols in the code string $C(s)$, which may be either transmitted or stored in an appropriate storing device. We may imagine this device to form an extension of the register C to the left. Hence, the code $C(s)$ consists of all such symbols as well as those residing inside the register C. As in the binary case the carry over problem may be handled by the bit stuffing technique.

We illustrate the encoding algorithm by a simple example. Let $n = 1000$, $n_0 = 0010$, $n_1 = 0011$, and $n_2 = 0011$. After normalization, we have $\bar{n} = 1.000$, $n_0 = 0.010$, $n_1 = 0.011$, $n_2 = 0.011$, so that $\bar{Q}(0) = 0.000$, $\bar{Q}(1) = 0.010$, and $\bar{Q}(2) = 0.101$. We will use $w = 4$. Let the string s be 100212. Table III lists the contents of both registers after each symbol has been encoded.

The decoding unit uses the same two registers as the encoding unit with the same interpretation of their contents as binary fractional numbers. The decoding is done with the following algorithm.

B. Decoding Algorithm

Fill C with w first symbols of the code string and set A to $10 \cdots 0$.

1) Read $\bar{Q}(m)$ for the next symbol and test if $\bar{Q}(m) < A$. If

TABLE III
ENCODING EXAMPLE FOR NONBINARY ALPHABET

Symbol	*Reg C*	*Reg A*
Start	0000	1000
1	0 1000	0110
0	010 0000	1000
0	01000 0000	1000
2	010001 0100	0110
1	0100011 0000	0110
2	0100011101 0000	1000

TABLE IV
DECODING EXAMPLE FOR NONBINARY ALPHABET

Decoded Symbol	*Reg C*	*Reg A*
Start	0100	1000
1	0000	0110
0	0011	1000
0	1110	1000
2	1001	0110
1	1010	0110
2	0000	1000

true, put $\beta = 0$. Otherwise, put $\beta = 1$. Then find the largest symbol j such that $2^{-\beta}\bar{Q}(j) \leq C$. Decode the symbol as the so-found largest symbol, say i.

2) If $i < m$, subtract $2^{-\beta}\bar{Q}(i)$ from C and insert $2^{-\beta}\bar{n}_i$ in register A. Go to step 4).

3) If $i = m$, subtract $2^{-\beta}\bar{Q}(m)$ from both C and A.

4) Shift the contents of both registers C and A left as many positions as needed to bring the content of A to the range [0.75, 1.5). Fill the remaining positions in C with new symbols from the code string. If none exists, stop. Else, go to Step 1).

We illustrate the decoding algorithm by decoding the code stream generated by the encoder of our example above. Table IV lists the contents of both registers after each symbol has been decoded.

V. Code Efficiency

In this section, we analyze the code efficiency, and discuss the choice of certain parameters, such as the optimum scaling range. For the convenience of analysis, we assume that the input parameters are available as probabilities rather than counts. Since it is crucial to the understanding of the analysis of code efficiency, we repeat here the update rule for $a(s)$. After a string s^t has been encoded, the update rule for $a(s)$ when the next symbol x_{t+1} is observed is the following:

$$a(s^t x_{t+1}) = \begin{cases} \dfrac{p(x_{t+1}|s^t)}{k(t)} & \text{if } x_{t+1} \neq m \\ a(s^t) - \dfrac{(1-p_m)}{k(t)} & \text{if } x_{t+1} = m \end{cases} \tag{5.1}$$

where $k_t = 1$, if $a(s^t) > (1 - p_m)$, and $k_t = 2$, if $a(s^t) \leq (1 - p_m)$, m is the last symbol in the alphabet, and p_m is the probability assigned to the last symbol m. The most probable symbol in the alphabet should be chosen as the last symbol m for maximizing code efficiency, as will be shown later in this section. Please note that all logarithms in this section are to the base 2. Let L_a be the actual code length and L_i the ideal code length after N symbols are encoded; let l_a be the corresponding actual per-symbol code length and l_i the ideal per-symbol code length. Then

$$L_a = -\sum_{t=0}^{N-1} \log \frac{a(s^t x_{t+1})}{a(s^t)}$$

$$= -\sum_{t=0}^{N-1} \log \frac{p(x_{t+1}|s^t)}{k(t)a(s^t)} + C(p_m)$$

where $C(p_m)$ is a correction term due to the different way we update $a(s)$ when the next symbol happens to be m, and can be expressed as

$$C(p_m) = -\sum_{t:x_{t+1}=m}^{N-1} \log \left(a(s^t) - \frac{(1-p_m)}{k(t)} \right) - \log \frac{p_m}{k(t)}. \tag{5.2}$$

Further,

$$L_a = -\sum_{t=0}^{N-1} \log p(x_{t+1}|s^t) + \sum_{t=0}^{N-1} \log (k(t)a(s^t)) + C(p_m)$$

$$= L_i + \sum_{t=0}^{N-1} \log (k(t)a(s^t)) + C(p_m).$$

Dividing both sides by N, the total number of symbols seen so far, we get the excess per-symbol code length over the idea code length as

$$\delta = l_a - l_i = \frac{1}{N}\sum_{t=0}^{N-1} \log (k(t)a(s^t)) + \frac{1}{N} C(p_m). \tag{5.3}$$

Now, for large values of N,

$$\frac{1}{N}\sum_{t=0}^{N-1} \log (k(t)a(s^t)) \cong E(\log (k(t)a(s^t)))$$

where $E(.)$ denotes the expected value. Let us assume that $a(s^t)$ forms a stationary process, uniformly distributed in its scaling range which we choose as [0.75, 1.5) for the present analysis. That is,

$$p(a(s^t)) = p(a) = \begin{cases} \dfrac{1}{0.75}, & 0.75 \leq a < 1.5 \\ 0, & \text{otherwise.} \end{cases}$$

Further, noting that $k(t) = 1$ for $a(s^t) > (1 - p_m)$ and $k(t) = 2$ for $a(s^t) \leq (1 - p_m)$, we get the first term in (5.3) as

$$\frac{1}{N}\sum_{t=0}^{N-1} \log k(t)a(s^t)$$

$$\cong \frac{1}{0.75}\left(\int_{0.75}^{1-p_m} \log 2a \, da + \int_{1-p_m}^{1.5} \log a \, da \right)$$

$$= 0.142267 + \frac{(1-p_m) - 0.75}{0.75}. \tag{5.4}$$

For the second term $1/N\,C(p_m)$, by (5.2) we have

$$\frac{1}{N}C(p_m) \cong -p_m\left(E\left(\log\left(a\left(s^t-\frac{(1-p_m)}{k(t)}\right)\right)\right)-E\left(\log\frac{p_m}{k(t)}\right)\right).$$

Again, with the uniform distribution for $a(s^t)$, we get

$$\begin{aligned}\frac{1}{N}C(p_m) &\cong -\frac{p_m}{0.75}\left(\int_{0.75}^{1-p_m}\log\left(a-\frac{(1-p_m)}{2}\right)da\right.\\ &\quad\left.+\int_{1-p_m}^{1.5}\log\,(a-(1-p_m))\,da\right)\\ &\quad+p_m\log p_m-p_m p(a(s^t)\le 1-p_m)\\ &=-\frac{p_m}{1.5}\left((1-p_m)\log\frac{(1-p_m)}{2e}\right.\\ &\quad\left.+(0.5+p_m)\log\frac{(1+2p_m)}{e}\right)+p_m\log p_m\\ &\quad-p_m\frac{(1-p_m)-0.75}{0.75}\end{aligned}\tag{5.5}$$

where e denotes the base of the natural logarithm. Combining (5.3), (5.4), and (5.5), we get the excess per-symbol code length as

$$\begin{aligned}\delta &= 0.142267+(1-p_m)\frac{(1-p_m)-0.75}{0.75}+p_m\log p_m\\ &\quad-p_m\left(\frac{1-p_m}{1.5}\log\frac{1-p_m}{2e}+\frac{0.5+p_m}{1.5}\log\frac{1+2p_m}{e}\right)\end{aligned}\tag{5.6}$$

and the efficiency of compression as

$$\text{Efficiency}=\frac{l_i}{l_a}=\frac{l_i}{l_i+\delta}.\tag{5.7}$$

In English text, the most probable symbol is usually found to be the letter "t". Suppose that we make "t" the last symbol m, in the compression of files containing English text. We found by modeling a number of such files as a stationary independent process that $p_m = 0.0624$, and the ideal per-symbol code length $l_i = 4.7887$ bits. From (5.6) we can calculate the excess code length δ to be 0.2556 bits per symbol, and from (5.7) the efficiency to be 94.4 percent. This agrees well with the experimental results in Table VI. Any deviations from experimental results are due to the simplifying assumption that $a(s^t)$ is uniformly distributed. For the above analysis, we assumed that $0.75 \le (1 - p_m) < 1.5$, as that is the general case. For large values of p_m, it is possible to have $0.75 > (1 - p_m)$, and the analysis of code efficiency turns out to be simpler because $k(t) = 1$ in (5.1).

In (5.6), the excess code length δ is dependent on p_m. It turns out that as p_m increases, δ decreases, which implies high efficiency. The decrease of δ with increasing p_m can be seen from entries in Table V, under the column $\delta(0.75)$. That is why it is recommended that the most probable symbol be considered as the last symbol in the alphabet.

A. Optimum Scaling Range

For the analysis in the previous section, we had assumed that $a(s)$ is scaled to be in the range [0.75, 1.5). In this section, we address the question of the optimum scaling range for maximizing the efficiency. Let the scaling range be $[k, 2k)$, and $k \le (1 - p_m) < 2k$. We can write a general expression for the excess code length in terms of k as

$$\begin{aligned}\delta(k) &= (1-p_m)\frac{(1-p_m)-k}{k}+2+\log\frac{k}{e}+p_m\log p_m\\ &\quad-\frac{p_m}{2k}\left((1-p_m)\log\frac{(1-p_m)}{e}+(2k-(1-p_m))\right.\\ &\quad\left.\cdot\log\frac{2(2k-(1-p_m))}{e}\right).\end{aligned}\tag{5.8}$$

TABLE V
OPTIMUM SCALING RANGE

p_m	k_{opt}	$\delta(k_{opt})$	$\delta(0.75)$	$\delta(0.5)$
0.02	0.6840	0.3662	0.3718	0.4361
0.04	0.6700	0.3018	0.3093	0.3589
0.06	0.6671	0.2529	0.2607	0.2999
0.08	0.6598	0.2144	0.2230	0.2533
0.10	0.6535	0.1836	0.1929	0.2162
0.12	0.6483	0.1592	0.1688	0.1868
0.14	0.6444	0.1400	0.1497	0.1639
0.16	0.6422	0.1253	0.1348	0.1466
0.18	0.6418	0.1145	0.1235	0.1340
0.20	0.6435	0.1072	0.1154	0.1258
0.22	0.6475	0.1029	0.1101	0.1213
0.24	0.6538	0.1013	0.1072	0.1190
0.26	0.6627	0.1019	0.1066	0.1220
0.28	0.6740	0.1045	0.1078	0.1265
0.30	0.6878	0.1087	0.1108	0.1334

This equation can be numerically solved for the optimum value k_{opt}, which is dependent on p_m. Table V lists k_{opt} for various values of p_m. Table V also gives the corresponding excess code length $\delta(k_{opt})$. k_{opt} varies from 0.6418 for $p_m = 0.18$–0.6878 for $p_m = 0.30$. However, for ease of hardware implementation it is desirable to keep the scaling range fixed. We have the option of keeping k fixed, for example, at 0.5, or, 0.75, etc. Table V gives the corresponding excess code lengths under $\delta(0.5)$ and $\delta(0.75)$. It can be seen that $\delta(0.75)$ is much closer to $\delta(k_{opt})$, than $\delta(0.5)$. Although we could consider other scaling ranges such as [0.625, 1.25), it will make the hardware more complex for little improvement in coding efficiency. Hence, we recommend using a scaling range of [0.75, 1.5). For $p_m > 0.3$, the solution of (5.8) leads to values of k_{opt} greater than $(1 - p_m)$. Since it violates the condition for the derivation of (5.8), those values are not valid, and hence we have not listed them. However, it is easy to derive an expression for $\delta(k)$ under the condition $k < (1 - p_m)$ and to compute k_{opt} from the resulting expression. Fortuitously, it turns out that the resulting values of k_{opt} ranges from 0.7472 for $p_m = 0.36$ to 0.70 for $p_m = 0.8$. Hence, our choice of $k = 0.75$ is indeed close to the optimum for most cases.

The efficiency of the code for binary alphabets tends to be higher because in such cases $p_m > 0.5$, which implies $a(s^t) > (1 - p_m)$ and $k(t) = 1$ in (5.1). There is a peculiar singularity in the performance of the binary code at the exact probability one half of the lower normalized bound of the A register, namely, 0.375 for one of the symbols, which gives by far the worst case performance. Indeed, with this probability the register A will always be normalized to the same value 0.75, which implies that a shift of one position occurs for each symbol occurrence. Hence, the code efficiency at this point is $H(0.375) \cong 0.955$ where $H(p)$ denotes the binary entropy function. The efficiency increases very rapidly to about 0.99 when the lower probability deviates from this value. Fortunately, this worst case performance does not occur in practice

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.