Block Arithmetic Coding for Source Compression

Charles G. Boncelet Jr., *Member IEEE*

***Abstract*— We introduce "Block Arithmetic Coding" (BAC), a technique for entropy coding that combines many of the advantages of ordinary stream arithmetic coding with the simplicity of block codes. The code is variable length in to fixed out (V to F), unlike Huffman coding which is fixed in to variable out (F to V). We develop two versions of the coder: 1) an optimal encoder based on dynamic programming arguments, and 2) a suboptimal heuristic based on arithmetic coding. The optimal coder is optimal over all V to F complete and proper block codes. We show that the suboptimal coder achieves compression that is within a constant of a perfect entropy coder for independent and identically distributed inputs. BAC is easily implemented, even with large codebooks, because the algorithms for coding and decoding are regular. For instance, codebooks with 2^{32} entries are feasible. BAC also does not suffer catastrophic failure in the presence of channel errors. Decoding errors are confined to the block in question. The encoding is in practice reasonably efficient. With i.i.d. binary inputs with $P(1) = 0.95$ and 16 bit codes, entropy arguments indicate at most 55.8 bits can be encoded; the BAC heuristic achieves 53.0 and the optimal BAC achieves 53.5. Finally, BAC appears to be much faster than ordinary arithmetic coding.**

***Index Terms*—Arithmetic codes, block codes, variable to fixed codes, entropy compression.**

I. Introduction

We introduce a method of source coding called Block Arithmetic Coding (BAC). BAC is a variable in to fixed out block coder (V to F) unlike Huffman codes which are fixed in to variable out (F to V). BAC is simple to implement, efficient, and usable in a wide variety of applications. Furthermore, BAC codes do not suffer catastrophic failure in the presence of channel errors. Decoding errors are confined to the block in question.

Consider the input, $X = x_1x_2x_3\ldots$ to be a sequence of independent and identically distributed input symbols taken from some input alphabet $A = \{a_1, a_2, \ldots, a_m\}$. The symbols obey probabilities $p_j = \Pr(x_l = a_j)$. Assume that $p_j > 0$ for all j. The entropy of the source is

$$H(X) = -\sum_{l=1}^{m} p_l \log p_l. \tag{1}$$

(We will measure all entropies in bits.) The compression efficiency of entropy encoders is bounded below by the entropy. For F to V encoders, each input symbol requires on average at least $H(X)$ output bits; for V to F encoders, each output bit can on average represent at most $1/H(X)$ input symbols.

The most important entropy compressor is Huffman coding [9]. Huffman codes are F to V block codes. They block the input into strings of q letters and encode these strings with variable length output strings. It is well-known that the number of output bits of a Huffman code is bounded above by $qH(X)+1$ for all block sizes. In special cases, this bound can be tightened [5], [17]. As a result the relative efficiency of Huffman codes can be made as high as desired by taking the block size to be large enough. In practice, however, large block sizes are difficult to implement. Codebooks are usually stored and must be searched for each input string. This storage and searching limits block sizes to small values. (However, see Jakobsson [11].)

Huffman codes can be implemented adaptively, but it is difficult to do so. The process of generating Huffman code trees is sufficiently cumbersome that adapting the tree on a symbol by symbol basis is rarely computationally efficient.

In recent years, a class of entropy coders called arithmetic coders has been studied and developed [8], [13], [18]–[21], [25]. Arithmetic coders are stream coders. They take in an arbitrarily long input and output a corresponding output stream. Arithmetic coders have many advantages. On average, the ratio of output length to input length can be very close to the source entropy. The input probabilities can be changed on a symbol by symbol basis and can be estimated adaptively. However, arithmetic codes have certain disadvantages. In some applications, the encoding and decoding steps are too complicated to be done in real time.

The entropy coder most similar to BAC is due to Tunstall [23], attributed in [12]. Tunstall's encoder is a V to F block coder that operates as follows: Given a codebook with $K - m + 1$ output symbols, a codebook with K output symbols is generated by "splitting" the most probable input sequence into m successors, each formed by appending one of the a_j. Unfortunately, the encoding of input sequences appears to require searching a codebook. Thus, large block sizes appear to be prohibitive, both in creating the code and in searching it. This has the practical effect of limiting block sizes to small values.

Recently, a V to F block arithmetic coder has been proposed by Teuhola and Raita [22]. While similar in spirit to this work, their coder, named FIXARI, differs in many details from BAC. FIXARI parses the input differently from BAC and encodes each string differently. The analysis of FIXARI is entirely different from the analysis of BAC presented in Section III. FIXARI does appear to be fast.

Manuscript received September 9, 1991; revised February 15, 1993. This paper was presented in part in The 1992 Conference on Information Sciences and Systems, Princeton, NJ, March 1992. This work was performed in part while the author was visiting at the University of Michigan.

The author is with the Department of Electrical Engineering, University of Delaware, Newark, DE 19716.

IEEE Log Number 9211435.

Other V to F entropy coders are runlength [7] and Ziv-Lempel [14], [24], [27]. Runlength codes count the runs of single symbols and encode the runs by transmitting the number in each run. They work well when the probabilities are highly skewed or when there is significant positive correlation between symbols. Runlength codes are often used for small alphabets, especially binary inputs. Then the runlength code is particularly simple since, by necessity, the runs alternate between 1's and 0's. Runlength codes are occasionally combined with Huffman to form a variable in to variable out (VtoV) scheme. For example, consider the CCITT facsimile code [10] or the more general VtoV runlength codes of Elias [6].

Ziv-Lempel codes work on different principles than BAC codes or the other codes considered so far. They find substrings of input symbols that appear frequently and encode each with a fixed length codeword. Ziv-Lempel codes work particularly well on English text, as they can discover complex dependencies between the symbols.

One advantage of most fixed output length block coders, including BAC, is that the effects of transmission errors (symbols received incorrectly) are confined to the block in question. The decoder does not lose synchronism with the incoming bit stream. While the corrupted block may be decoded incorrectly, remaining blocks will be decoded correctly. However, there may be insertions or deletions in the decoded stream. These insertions and deletions can usually be dealt with at the application level. In contrast, Huffman codes and arithmetic codes can lose synchronism and suffer catastrophic failure in the presence of channel errors. (It is often argued that Huffman codes are self-synchronizing, but this is not guaranteed. Various techniques have been proposed to combat this problem. See, e.g., [16].)

One subtle point is that fixed output length codes can still suffer catastrophic failure if implemented in an adaptive fashion. If the probability estimates depend on previously transmitted symbols, then a single error can affect other blocks. In particular, Ziv-Lempel codes are usually implemented in an adaptive fashion and are susceptible to catastrophic failure if their dictionaries get corrupted. BAC codes will be vulnerable to the same catastrophic failure if the probabilities are estimated adaptively based on prior symbols.

Finally, there is a growing recognition that V to F codes can outperform F to V codes with strong dependencies between symbols [15], [26]. The idea is that, for a fixed number of symbols, V to F codes can match long input strings. For example, a runlength coder with K codewords can match a run of up to $K-1$ symbols. This run can be encoded with approximately $\log K$ bits, resulting in an exponential gain. In contrast, Huffman codes only give arithmetic gains. For K large enough, the V to F coder is more efficient.

This paper is organized as follows: Section II develops the optimal and heuristic BAC's. In Section III, we present the principal theoretical result of this paper: that BAC's are within an additive constant of the entropy for all code sizes. Section IV presents computations illustrating the efficiency of BAC's. Section V discusses extensions to time varying and Markov symbols, Section VI discusses the EOF problem and proposes two new solutions, and Section VII presents conclusions and discussion.

II. Development Of Block Arithmetic Coding

The BAC encoder parses the input into variable length input strings and encodes each with single fixed length output codewords.

The following definitions are taken from Jelinek and Schneider [12]: Let $W(K) = \{w_i : 1 \le i \le K\}$ be a K element collection of words, w_i, with each word a substring of X. $W(K)$ is *proper* if, for any two words, w_i and w_j with $i \neq j$, w_i is not a prefix of w_j. $W(K)$ is *complete* if every infinite length input string has a prefix in W. The characterization in [15] is succinct: a code is complete and proper if every infinite length input string has one and only one prefix in $W(K)$.

It is useful to think of complete and proper from the viewpoint of parsing trees. *Proper* means that the codewords are all leafs of the tree; *complete* means that all the nodes in the tree are either leafs or they are internal nodes with m children.

The number of codewords in complete and proper sets is $1 + L(m-1)$ for $L = 1, 2, \ldots$ [12]. This result is also easy to derive from the viewpoint of trees. The root has m children ($L = 1$). Each time a leaf node is replaced by m children, a net gain of $m-1$ leafs result. Below, we will allow $L = 0$ so that one codeword can be allowed in appropriate sets.

We assume that to each output codeword is assigned an integer index i with $i = 1, 2, \ldots, K_S$. Let S be the set of output codewords. Consider a subset of codewords with contiguous indices. Denote the first index by A and the last by B. The number of codewords in the subset is $K = B - A + 1$.

BAC proceeds by recursively splitting S into disjoint subsets. With each input symbol, the current subset is split into m nonempty, disjoint subsets, one for each possible input letter. The new subset corresponding to the actual letter is chosen and the process continues recursively. When the subset has fewer than m codewords, BAC stops splitting, outputs any of the codewords in the subset, reinitializes, and continues. The ideal situation is that each final subset contain only 1 codeword; otherwise, the "extra" codewords are wasted.

Denote the number of codewords in each of the m disjoint subsets by $K_l(K)$, where K is the number of codewords in the current set. Usually, we will suppress the dependence on K and denote the number simply by K_l.

The question of how the K_l should be selected is central to the remainder of this paper. We can identify five criteria that are necessary or desirable:

1) $K_l > 0$.
2) $\sum_{l=1}^{m} K_l \le K$.
3) $\sum_{l=1}^{m} K_l = K$.
4) $K_l = 1 + L_l(m-1)$ for some $L_l = 0, 1, 2, \ldots$.
5) If $p_j > p_i$, then $K_j \ge K_i$.

Criteria Q1 and Q2 are necessary to ensure that the parsing is complete and proper and that the encoding can be decoded. Each subset must have at least one codeword and the sum of all the subsets cannot exceed the total available.

Criteria Q3, Q4, and Q5 are desirable in that meeting each results in a more efficient encoding. In effect, Q3 says that one ought to use all the codewords that are available. Q4 assures that the number of codewords in each subset is equal to one (a leaf) or is the number in a complete and proper set. For

```
/* BAC Encoder */
$K = K_S$;
$A = 1$;
while (($l$ = getinput()) $\neq$ EOF)
    {
    Compute $K_1, K_2, \ldots, K_m$;
    $A = A + \sum_{i=1}^{l-1} K_i$;
    $K = K_l$;
    if ($K < m$)
        {
        Output code($A$);
        $A = 1$;
        $K = K_S$;
        }
    }
doeof($A, K$);
```

Fig. 1. Basic BAC Encoder.

```
/* BAC Decoder */
while (($C$ = getindexcode()) $\neq$ EOF)
    {
    $K = K_S$;
    $A = 1$;
    while ($K \geq m$)
        {
        Compute $K_1, K_2, \ldots, K_m$;
        Find $l$ s.t. $A + \sum_{i=1}^{l-1} K_i \leq C < A + \sum_{i=1}^{l} K_i$;
        Output $a_l$;
        $A = A + \sum_{i=1}^{l-1} K_i$;
        $K = K_l$;
        }
    }
undoeof($A, K$);
```

Fig. 2. Basic BAC Decoder

example, consider $m = 3$ and $K = 5$. The subsets should be divided into some permutation of $1, 1, 3$, and not $1, 2, 2$. In the former case, the set with 3 elements can be divided one more time; in the latter case, none of the subsets can. Note, Q3 and Q4 can both be satisfied if the initial K satisfies $K = 1 + L(m-1)$ and $\sum_{l=1}^{m} L_l = L - 1$. Q5 merely reflects the intuitive goal that more probable letters should get more codewords. In theory, Q5 can always be met by sorting the K_l's in the same order as the p_l's. Sometimes in practice, e.g., in adaptive models, it may be computationally difficult to know the sorted order and meeting Q5 may be problematic.

The BAC encoding algorithm is given in Fig. 1. The BAC decoding algorithm is given in Fig. 2.

We assume that the various functions behave as follows:

[getinput()] Returns the index of the next input symbol to be coded. If a_r is the next letter, then it returns r.

[code(A)] Returns the codeword corresponding to codeword index, A.

[doeof(A, K)] Handles input end-of-file (EOF). Discussed below in Section VI.

[getindexcode()] Returns the index of the next codeword taken from the channel.

[undoeof()] Undoes the effect of doeof().

As examples of the kinds of parsings available, consider the following for binary inputs ($a_1 = 0, a_2 = 1$):

- Let $K_1 = K - 1$ and $K_2 = 1$. These are runlength codes for runs of 0's. Similarly, if $K_1 = 1$ and $K_2 = K - 1$, we get runlength codes for runs of 1's. The parsing trees are as much "left" or "right" as possible.
- Let $K_1(K_S) = K_S/2$ and $K_2(K_S) = K_S/2$. If the first symbol is a 1, then let $K_1 = K - 1$ and $K_2 = 1$ for all other K's; else, let $K_1 = 1$ and $K_2 = K - 1$. These are symmetric runlength codes, useful when the starting value is unknown.
- Let $K_1 = K/2$ and $K_2 = K/2$. These codes map the input to output in an identity-like fashion.

With the conditions above, we can state the following theorem:

Theorem 1 Block Arithmetic Codes are complete and proper. Furthermore, all complete and proper parsings can be developed as BAC's.

Proof: It is easiest to develop this equivalence with parsing trees. BAC codes are complete and proper from their construction. They are proper because no outputs are generated at nonleaf nodes; they are complete because each terminal node has $m - 1$ siblings. For the converse, consider any complete and proper parsing as a tree. At each node of the tree, a BAC coder can reproduce the same tree by appropriately choosing the subset sizes, K_l. For instance, at the root node, count the number of leafs in each branch. Use each of these numbers for K_l, respectively. The process can clearly continue at each node. □

In the case considered so far of independent and identically distributed input symbols, it is straightforward to derive an expression for the efficiency of a code. Let $N(K)$ denote the expected number of input symbols encoded with K output symbols. Due to the recursive decomposition, one obtains the following recursive formula:

$$N(K) = 1 + \sum_{l=1}^{m} p_l N(K_l), \tag{2}$$

The "1" is for the current symbol, x_j. The sum follows from considering all possible choices for x. With probability p_l, the input is a_l and the subset chosen has K_l codewords. The expected number of input symbols encoded from here onward is $N(K_l)$.

Since this equation is crucial to the rest of this development, we also present an alternate derivation. $N(K)$, as the expected number of input symbols encoded with K output symbols, can be written as

$$N(K) = \sum_{i=1}^{K} n_i \Pr(w_i),$$

where n_i is the number of bits in w_i. Parse $w_i = a_{1,i} w_i'$ where $a_{1,i}$ is the first letter in w_i. Then $n_i' = n_i - 1$ and, by independence, $\Pr(w_i) = \Pr(a_{1,i})\Pr(w_i')$. Thus,

$$\begin{aligned} N(K) &= \sum_{i=1}^{K}(1+n_i')\Pr(a_{1,i})\Pr(w_i') \\ &= 1+\sum_{i=1}^{K} n_i' \Pr(a_{1,i})\Pr(w_i') \\ &= 1+\sum_{l=1}^{m} p_l \sum_{\{i:a_{1,i}=a_l\}} n_i' \Pr(w_i') \\ &= 1+\sum_{l=1}^{m} p_l N(K_l), \end{aligned} \tag{3}$$

(3) follows because K_l codewords start with a_l.

The boundary conditions are easy and follow from the simple requirement that at least m codewords are needed to encode a single symbol:

$$N(K) = 0 \quad K = 0, 1, \ldots, m-1. \tag{4}$$

Note, one application of (2) yields $N(m) = 1$. While hard to compute $N(K)$ in closed form, except in special cases, (2) and (4) are easy to program.

For an example of an easy case, consider ordinary runlength coding for binary inputs. Letting $N_r(K)$ refer to the expected number of input symbols encoded with K output codewords using runlength encoding, then the recursion simplifies to

$$\begin{aligned} N_r(K) &= 1 + pN_r(K-1) + qN_r(1) \\ &= 1 + pN_r(K-1), \end{aligned} \tag{5}$$

where p is the probability of a symbol in the current run and $q = 1 - p$. (5) follows since $N_r(1) = 0$. The solution to this linear difference equation is

$$N_r(K) = (1 - p^{K-1})/(1-p). \tag{6}$$

Interestingly, this solution asymptotically approaches $1/(1-p)$. We arrive at the well-known conclusion that V to F runlength codes do not work well for large block sizes.

We propose two specific rules for determining the size of each subset. The first is an optimal dynamic programming approach, letting $N_o(K)$ be the optimal expected number of input symbols encoded using K output codewords:

$$N_o(K) = 1 + \max_{K_1, K_2, \ldots, K_m} \sum_{l=1}^{m} p_l N_o(K_l), \tag{7}$$

subject to the constraints $K_l \geq 1$ and $\sum_{l=1}^{m} K_l = K$. This optimization is readily solved by dynamic programming since all the K_l obey $K_l < K$. However, if m is large, a full search may be computationally prohibitive. The optimal solution can be stored in tabular form and implemented with a table lookup. Note, the optimal codebook does not have to be stored, only the optimal subset sizes.

The second is a heuristic based on ordinary arithmetic coding. In arithmetic coding, an interval is subdivided into disjoint subintervals with the size of each subinterval proportional to the probability of the corresponding symbol. We propose to do the same for BAC. K_l should be chosen as close to $p_l K$ as possible. Let $K_l = [p_l K]$, where $[s]$ is the quantization of s, with the proviso that each $K_l \geq 1$.

```
/* Good Heuristic to determine the L's ($K_l = 1 + L_l(m-1)$) */
Sort the symbols so that $p_1 \leq p_2 \leq \cdots \leq p_m$.
$q = 1.0$;
$\tilde{L} = L$;
for ($l = 1$; $l \leq m$; $l$++)
    {
    $\tilde{p} = p_l/q$;
    $L_l = \lfloor \tilde{p}\tilde{L} + ((\tilde{p}(2-l)-1)/(m-1)+0.5) \rfloor$;
    if ($L_l < 0$)
        $L_l = 0$;
    $\tilde{L} = \tilde{L} - L_l$;
    $q = q - p_l$.
    }
```

Fig. 3. "Good" probability quantizer. Computes L_l for $l = 1, 2, \ldots, m$ given L. Note, all quantities not involving L can be precomputed.

One way to do the quantization consistent with Q1–Q4 above is described in Fig. 3. The idea is to process the input letters from least probable to most probable and, for each letter, to keep track of the number of codewords remaining and the total probability remaining. The algorithm actually computes L_l, not K_l, as the former is somewhat easier. Note, Q5 may not be satisfied due to quantization effects. However, our computations and simulations indicate that it almost always is. Furthermore, Q5 can be satisfied if desired by sorting the L_l's.

One drawback of the "Good" quantizer in Fig. 3 is that the L_l's are computed one at a time from the least probable letter up to the letter input. Potentially, this loop may execute m times for each input symbol. For large m, this may be too slow. One way to circumvent this problem is the "Fast" quantizer described below. The idea behind the fast quantizer is to compute a "cumulative L function", R_l, or, equivalently, a "cumulative K function", Q_l, and form either L_l or $K_l = 1 + (m-1)L_l$ by taking a difference. The equations for R and L take the form:

$$R_0 = 0 \tag{8}$$

$$R_l = [(L-1)\sum_{j=1}^{l} p_j] \tag{9}$$

$$L_l = R_l - R_{l-1} \tag{10}$$

Those for Q and K are as follows:

$$Q_0 = 0 \tag{11}$$

$$Q_l = l + (m-1)R_l \tag{12}$$

$$K_l = Q_l - Q_{l-1} \tag{13}$$

The "Fast" quantizer satisfies Q1–Q4, though not necessarily Q5. For large alphabets it is slightly less efficient than the "Good" quantizer, but is much faster because there is no loop over all the symbols.

The algorithmic complexity of BAC can be summarized as follows: Using the good heuristic, the encoder can require up to $m - 1$ multiplications per input symbol, and a similar number of additions; using the fast heuristic, it only requires two multiplications and a fixed number of additions and comparisons per input symbol. The decoder does the same computations of K_l (or L_l) as the encoder, but also has to search for the proper interval. This searching can be done in $O(\log m)$ operations. Thus, the encoder can be implemented in $O(1)$ or $O(m)$ operations per input symbol and the decoder in $O(\log m)$ or $O(m)$ operations, depending on whether the fast or the good heuristic is used. In the special case of binary inputs, there is little difference, both in complexity and performance, between the two heuristics. After the K_l's have been computed, the optimal BAC can be implemented with no multiplications and a similar number of additions to the fast heuristic. However, the one time cost of computing the optimal K_l's may be high (a brute force search requires approximately $O(m^{L_S})$ operations if m is large, where $K_S = 1+L_S(m-1)$).

In some applications, it is convenient to combine the optimal and heuristic approaches in one hybrid approach: use the optimal sizes for small values of K and use the heuristic for large values. As we argue in Section IV, both encoders incur their greatest inefficiency for small values of K. For instance, an encoder can begin with the heuristic until 256–1024 codewords are left and then switch to the optimal.

Letting $N_h(K)$ represent the expected number of input symbols encoded with K output codewords under the heuristic, the expression for $N(K)$ becomes

$$N_h(K) = 1 + \sum_{l=1}^{m} p_l N_h([p_l K]). \tag{14}$$

For the moment, if we ignore the quantizing above, relax the boundary conditions, and consider $N(\cdot)$ to be a function of a continuous variable, say s, we get the following equation:

$$N_e(s) = 1 + \sum_{l=1}^{m} p_l N_e(p_l s), \tag{15}$$

where the subscript e refers to "entropy". The entropy rate for V to F codes,

$$N_e(s) = \frac{\log s}{H(X)}, \tag{16}$$

satisfies this equation.[1] Thus we see that, in the absence of quantization effects, BAC codes are entropy codes. As we argued above, for large K the quantization effects are small. Furthermore, $\log K$ is a slowly varying function of K. We might expect the heuristic to perform close to the optimal, entropy rate. In fact, we prove in Section III below that $N_h(K)$ is within an additive constant (which depends on the p's and m) of the entropy rate for all K.

III. Asymptotic Results

Consider the heuristic encoder discussed above. Assume the quantization is done so that, for K large enough, $[p_l K] = p_l K + \delta_l$, where $|\delta_l| \le m$. Assume further that $\sum_{l=1}^{m} \delta_l = 0$. (Such quantization can always be achieved for K large enough. See, e.g., Fig. 3.) In this section, we will take all "logs" to be natural logs to the base e.

[1] We believe this entropy solution uniquely solves (15), but have been unable to prove uniqueness.

Theorem 1 For independent and identically distributed inputs,

$$\log K - H(X)N_h(K) \le C, \tag{17}$$

for all K and some constant C. (C depends on the probabilities, the quantization, and on the size of the alphabet, but not on K.)

Proof: We will actually show a stronger intermediate result, namely that

$$\log K - H(X)N_h(K) \le G(K) = C - \frac{D}{K}, \tag{18}$$

where $D > 0$. Clearly $C - D/K \le C$.

The proof is inductive. There are three constants to be determined: C and D, and a splitting constant, K_C. First, select any K_C such that $K_C > 2m/p_1$, where $p_1 > 0$ is the minimum of the p_l's.

The basis is as follows: For all $K \le K_C$, and for any $D > 0$, we can choose C so that

$$\begin{aligned} C &\ge \log K - H(X)N_h(K) + \frac{D}{K} \\ &\ge \log K_C - 0 + \frac{D}{1}, \end{aligned} \tag{19}$$

where, clearly, $N_h(K) \ge 0$ and $D/K \le D$. At this point, K_C has been determined and C has been specified in terms of D and the proposition holds for all $K \le K_C$.

For the inductive step, we now assume that (18) applies for all $K \le K'$ for some $K' \ge K_C$. Then, we find a condition on D so that (18) applies for all K. Let $K = K' + 1$, then

$$\begin{aligned} &\log K - H(X)N_h(K) \\ &= H(X) + \sum_{l=1}^{m} p_l \log(p_l K) \\ &\quad - H(X) - \sum_{l=1}^{m} p_l H(X) N_h([p_l K]) \\ &= \sum_{l=1}^{m} p_l(\log(p_l K) - H(X)N_h([p_l K])) \\ &= \sum_{l=1}^{m} p_l(\log(p_l K) - \log([p_l K]) \\ &\qquad + \log([p_l K]) - H(X)N_h([p_l K])) \\ &\le \sum_{l=1}^{m} p_l(\log(p_l K) - \log([p_l K])) + \sum_{l=1}^{m} p_l G([p_l K]). \end{aligned} \tag{20}$$

The inductive hypothesis is used in (20). The last term above is easily bounded since $G(\cdot)$ is nondecreasing:

$$\sum_{l=1}^{m} p_l G([p_l K]) \le G(K-1). \tag{21}$$

Now consider each summand in the first term:

$$p_l(\log(p_l K) - \log([p_l K])) = p_l \int_{[p_l K]}^{p_l K} s^{-1} ds. \tag{22}$$

If $\delta_l \geq 0$, then

$$p_l \int_{[p_l K]}^{p_l K} s^{-1} ds \leq -\delta_l p_l \inf\{s^{-1} : p_l K \leq s \leq p_l K + \delta_l\} = \frac{-\delta_l p_l}{p_l K + \delta_l}. \tag{23}$$

Similarly, if $\delta_l < 0$,

$$p_l \int_{[p_l K]}^{p_l K} s^{-1} ds \leq -\delta_l p_l \sup\{s^{-1} : p_l K + \delta_l \leq s \leq p_l K\} = \frac{-\delta_l p_l}{p_l K + \delta_l}. \tag{24}$$

(By construction, $p_l K + \delta_l > 0$.)

The first term of (20) can be bounded as follows:

$$\begin{aligned}\sum_{l=1}^{m} & p_l(\log(p_l K) - \log([p_l K])) \\ &\leq -\sum_{l=1}^{m} \frac{\delta_l p_l}{p_l K + \delta_l} \\ &= -\sum_{l=1}^{m} \left(1 + \frac{\delta_l}{p_l K}\right)^{-1} \frac{\delta_l}{K} \\ &= -\left[\prod_{i=1}^{m}(1 + \frac{\delta_i}{p_i K})\right]^{-1} \sum_{l=1}^{m} \frac{\delta_l}{K} \prod_{j=1, j\neq l}^{m} \left(1 + \frac{\delta_l}{p_l K}\right) \\ &= -\left[\prod_{i=1}^{m}(1 + \frac{\delta_i}{p_i K})\right]^{-1} \sum_{l=1}^{m} \frac{\delta_l}{K} \left(1 + \sum_{j=1}^{m-1} \alpha_{lj} K^{-j}\right),\end{aligned} \tag{25}$$

where α_{lj} are coefficients that are polynomial functions of the δ_l's. They do not otherwise depend on K. Since $\sum_{l=1}^{m} \delta_l = 0$, we get the following:

$$\begin{aligned}\sum_{l=1}^{m} & p_l(\log(p_l K) - \log([p_l K])) \\ &\leq -\left[\prod_{i=1}^{m}(1 + \frac{\delta_i}{p_i K})\right]^{-1} K^{-1} \sum_{l=1}^{m} \delta_l \sum_{j=1}^{m-1} \alpha_{lj} K^{-j} \\ &= -\left[\prod_{i=1}^{m}(1 + \frac{\delta_i}{p_i K})\right]^{-1} K^{-1} \sum_{j=1}^{m-1} \beta_j K^{-j}.\end{aligned} \tag{26}$$

Where $\beta_j = \sum_{l=1}^{m} \delta_l \alpha_{lj}$ are polynomial functions of the δ_l's and do not otherwise depend on K. Now we can bound each term separately. By construction, $1 + \delta_i/(p_i K) > 1/2$. Then,

$$\left[\prod_{i=1}^{m}(1 + \frac{\delta_i}{p_i K})\right]^{-1} \leq 2^m. \tag{27}$$

The second term of (26) can be bounded as follows:

$$-\sum_{j=1}^{m-1} \beta_j K^{-j} \leq (m-1)K^{-1} \max_j |\beta_j|. \tag{28}$$

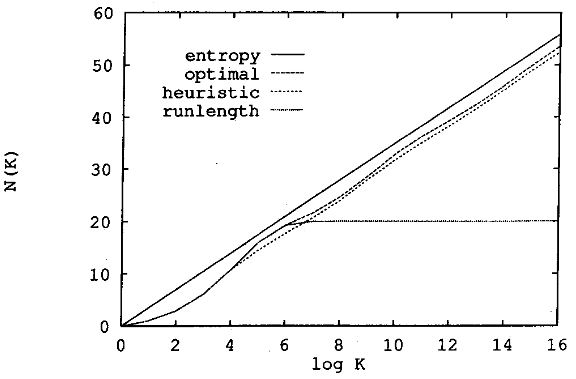

Fig. 4. Calculated $N(K)$ versus $\log K$ for binary inputs with $p = 0.95$.

Combining (21), (27), and (28) and letting $D' = (m-1)2^m \max_j |\beta_j|$, we get the following:

$$\log K - H(X)N_h(K) \leq G(K-1) + D'K^{-2} \leq G(K). \tag{29}$$

Substituting in for $G(\cdot)$, we need to select D so that the inequality in (29) is valid:

$$C - \frac{D}{K-1} + \frac{D'}{K^2} \leq C - \frac{D}{K}, \tag{30}$$

or, equivalently,

$$-DK + D'\frac{K-1}{K} \leq -DK + D, \tag{31}$$

which is valid for all $D > D'$. This completes the proof. □

Clearly, the optimal approach is better than the heuristic and worse than entropy. So, combining with the theorem above, we have the following ordering:

$$\frac{\log K}{H(X)} = N_e(K) \geq N_o(K) \geq N_h(K) \geq \frac{\log K - C}{H(X)}. \tag{32}$$

IV. Computational Results

In this section, we compute BAC efficiencies for the heuristic and optimal implementations and compare these with entropy values. We also compare an implementation of BAC and the arithmetic coder of Witten, Neal, and Cleary [25], both operating on a binary input, and show that BAC is much faster. Source code for the tests presented here can be found in [4].

In Fig. 4, we present calculations of $N(K)$ for a hypothetical entropy coder, the optimal BAC, the good heuristic BAC, and a simple V to F runlength coder versus $\log K = 0, 1, \ldots, 16$. The input is binary with $p = \Pr(x = 1) = 0.95$. Note that the optimal and heuristic curves closely follow the entropy line. The maximum difference between both curves and entropy is 4.4 and occurs at $\log K = 3$. In contrast, for $\log K = 16$, the optimal is within 2.3 input symbols of the entropy and the heuristic within 2.8. This behavior seems to be consistent over a wide range of values for p. The maximum difference occurs for a small value of $\log K$. The coders improve slightly in an absolute sense as $\log K$ grows. At some point the difference seems to level off. The relative efficiency increases as the number of codewords grow. At $\log K = 3$, it

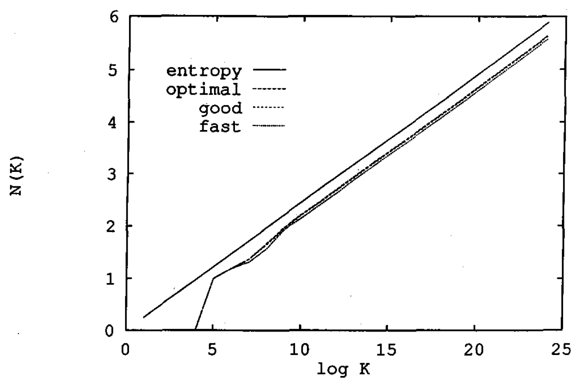

Fig. 5. Calculated $N(K)$ versus $\log K$ for a 27 letter english alphabet

TABLE I
CALCULATED $N(K)$ VERSUS $\log K$ FOR BINARY INPUTS WITH $p = 0.95$

	Entropy	Optimal		Heuristic		Runlength
p	$N_e(K)$	$N_o(K)$	$\tilde{C}$	$N_h(K)$	$\tilde{C}$	$N_R(K)$
0.80	22.2	21.9	0.22	21.9	0.22	5.0
0.85	26.2	25.8	0.24	25.7	0.30	6.7
0.90	34.1	33.2	0.42	33.1	0.47	10.0
0.95	55.9	53.5	0.69	53.1	0.80	20.0
0.98	113.1	103.4	1.37	101.8	1.60	50.0
0.99	198.0	184.9	1.13	181.2	1.36	100.0

is 58% for both and, at $\log K = 16$, it is 96% for the optimal BAC and 95% for the heuristic BAC.

In Fig. 5, we present calculations of $N(K)$ versus $\log K$ for a 27 letter English alphabet taken from Blahut [1, p. 21]. Plotted are $N(K)$ for the entropy, optimal, and good and fast heuristic BAC's. On this example, the curves for the optimal and the good heuristic are almost indistinguishable.

In Table I, we present calculated $N(K)$ for binary inputs and selected p's with $\log K = 16$, i.e., 16 bit codewords. One can see that both the heuristic and optimal BAC's are efficient across the whole range. For instance, with $p = 0.80$, both exceed 98.7% of entropy; with $p = 0.95$, both exceed 95.0%; and with $p = 0.99$, both exceed 91.5%.

The computational results support an approximation. For all K large enough, $H(X)N_h(K) \approx \log K - \tilde{C}$. In general, it seems to be that $\tilde{C} < C$, where C is taken from Theorem 2. Also in Table I are computed values of $\tilde{C} = \log K - H(X)N_h(K)$ for $K = 2^{16}$.

As another experiment, we implemented BAC and the coder of Witten, Neal, and Cleary (referred to as WNC) [25] to assess the relative speeds. Source code for WNC appears in their paper, so it makes a good comparison. Speed comparisons are admittedly tricky. To make the best comparison, both BAC and WNC were optimized for a binary input and the encoders for both share the same input routine and the decoders the same output routine. There are two versions of WNC, a straightforward C version, and an optimized C version. We implemented both and found that, after optimizing for binary inputs, the straightforward version was actually slightly faster than the optimized version. Both were tested on the same input file, 2^{20} independent and identically distributed bits with a

TABLE II
EXECUTION TIMES FOR BAC AND WNC FOR A 1 MILLION BIT FILE

Version	Time (secs)
BAC Encoder	4.2
BAC Decoder	3.7
WNC Encoder	21.6
WNC Decoder	29.5
WNC Encoder (fast)	22.2
WNC Decoder (fast)	29.8
IO Speed (encoder)	2.0
IO Speed (decoder)	1.7

probability of a 1 equal to 0.95. The execution times on a SPARC IPC are listed in Table II. The times are repeatable to within 0.1 seconds. Also listed are the times for input and output (IO Speed), i.e., to read in the input one bit at a time and write out the appropriate number of bytes, and to read in bytes and write out the appropriate number of bits. The IO Speed numbers do not reflect any computation, just the input and output necessary to both BAC and WNC.

We see that in this test, the BAC encoder is approximately 5 times faster than WNC and the BAC decoder is 8 times faster than the WNC decoder. Indeed, the BAC times are not much longer than the input/output operations alone.

V. EXTENSIONS

BAC's can be extended to work in more complicated environments than that of i.i.d. symbols considered so far. For instance, consider the symbols to be independent, but whose probabilities are time varying:

$$p(l, j) = \Pr(x_j = a_l). \tag{33}$$

Then denote the number of input symbols encoded with K output symbols starting at time j by $N(K, j)$. Then, in analogy to (2) we get the following:

$$N(K(j), j) = 1 + \sum_{l=1}^{m} p(l, j) N(K_l(j+1), j+1), \tag{34}$$

where $K_l(j)$ is the number of codewords assigned to a_l at time j. The heuristic is easy: Choose $K_l(j+1) = [p(l, j)K(j)]$. We can also present an optimal dynamic programming solution:

$$N_o(K(j), j) = 1 + \max_{K_1(j+1),\ldots,K_m(j+1)} \sum_{l=1}^{m} p(l, j) N_o(K_l(j+1), j+1). \tag{35}$$

This problem can be solved backward in time, from a maximum $j = K - m + 1$ to a minimum $j = 1$.

As another example consider a time invariant, first order Markov source. Let $p(i|l) = \Pr(x(j) = a_i | x(j-1) = a_l)$. Then the recursion for $N(K)$ splits into two parts. The first is for the first input symbol; the second is for all other input symbols:

$$N(K) = 1 + \sum_{l=1}^{m} p_l N(K_l | l) \tag{36}$$

$$N(K|l) = 1 + \sum_{i=1}^{m} p(i|l) N(K_i | i), \tag{37}$$

where $N(K|l)$ is the number of input symbols encoded using K codewords given that the current input is a_l. The heuristic again is easy: Choose $K_i = [p(i|l)K]$. The optimal solution is harder, although it can be done with dynamic programming. For every l, the optimal $N_o(K|l)$ can be found because each $K_i < K$. In [3], we show a similar optimality result to Theorem 2 for a class of first order Markov sources.

In some applications it is desirable to adaptively estimate the probabilities. As with stream arithmetic coding, BAC encodes the input in a first-in-first-out fashion. The only requirement is that the adaptive formula depends only on previous symbols, not on present or future symbols.

VI. The Eof Problem

One practical problem that BAC and other arithmetic coders have is denoting the end of the input sequence. In particular, the last codeword may be only partially selected.

The simplest solution to this problem is to count the number of symbols to be encoded and to send this count before encoding any. It is an easy matter for the decoder to count the number of symbols decoded and stop when the appropriate number is reached. However, this scheme requires that the encoder process the data twice and incurs a transmission overhead to send the count.

The EOF solution proposed in the stream arithmetic coder of Witten, Neal, and Cleary [25] and used in FIXARI [22] is to create an artificial letter with minimal probability. This letter is only transmitted once, denoting the end of the input. When the decoder decodes this special letter, it stops. We computed BAC's lost efficiency for binary inputs for a variety of probabilities and codebook sizes and found that it averages about 7%. For the English text example, the loss averaged about 3.5% over a wide range of codebook sizes.

We propose two alternatives to the simple schemes above (see also [2]). The first assumes the channel can tell the decoder that no more codewords remain and that the decoder is able to "lookahead" a modest number of bits. The idea is for the encoder to append to the input sequence as many least probable symbols as necessary (possibly zero) to flush out the last codeword. After transmitting the last codeword, the encoder transmits the number of encoded symbols to the decoder. Even with 2^{32} codewords, at most 31 extra input symbols are needed. This number can be encoded with 5 bits. The decoder looks ahead 5 bits until it detects that no more symbols are present. It then discards the appropriate number of appended symbols. The overhead caused by this scheme is modest: 5 bits at the end and one possibly wasteful codeword.

The second scheme assumes the channel can not tell the decoder that no more codewords remain. We suggest devoting one codeword to specify the end of the codeword sequence. (Note, we are not suggesting an extra input letter, but one of the K codewords.) Then append the extra number of bits as above. The inefficiency here includes the 5 bits above and the loss due to one less codeword. Using the approximation discussed in Section IV above, one computes the relative inefficiency as follows:

$$\frac{\log K - C' - \log(K-1) + C'}{\log K - C'} \approx (K(\log K - C'))^{-1} \quad (38)$$

For K large enough, this overhead is negligible.

VII. Conclusions And Discussion

We believe that BAC represents an interesting alternative in entropy coding. BAC is simple to implement, even for large block sizes, because the algorithm is top-down and regular. In contrast, Huffman's and Tunstall's algorithms are not top-down and generally require storing and searching a codebook.

BAC is efficient, though probably slightly less efficient than ordinary arithmetic coding. The comparison with Huffman is a little more complicated. The asymptotic result for BAC is, on the surface, weaker than for Huffman. The BAC bound uses C which must be computed on a case by case basis, while the Huffman bound is 1. Both coders can be made as relatively efficient as desired by selecting the block size large enough. However, BAC can use much larger block sizes than is practical for Huffman. The BAC asymptotic result is much stronger than that for Tunstall's coder. For BAC, the difference between entropy and the heuristic rates is bounded. Tunstall shows only that the ratio of rates of entropy and his coder approaches 1 as $K \to \infty$.

In the proof for Theorem 2, we have shown that a constant, C, exists that bounds the difference between entropy and the heuristic BAC for all K. We have made no effort to actually evaluate the constant from the conditions given in the proof. This is because such an evaluation would be worthless in evaluating the performance of BAC. As shown in Section IV, the constant in practice is quite small. One important need for future research is to provide tight bounds for C and, perhaps, to characterize the difference between the entropy rate and BAC more accurately.

One advantage of BAC compared to Huffman and stream arithmetic coding is that BAC uses fixed length output codewords. In the presence of channel errors, BAC will not suffer catastrophic failure. The other two might.

We have also argued that BAC can accommodate more complicated situations. Certainly the heuristic can handle time varying and Markov probabilities. It can estimate the probabilities adaptively. It remains for future work to prove optimality results for these more complicated situations.

References

[1] R. E. Blahut, *Principles and Practice of Information Theory.* Reading, MA: Addison-Wesley, 1987.

[2] C. G. Boncelet, Jr., "Extensions to block arithmetic coding," in *Proc. 1992 Conf.Inform. Sci. and Syst.*, Princeton NJ, Mar. 1992, pp. 691–695.

[3] ——, "Block arithmetic coding for Markov sources," in *Proc. 1993 Inform. Theory Symp.*, San Antonio, TX, Jan. 1993.

[4] ——, "Experiments in block arithmetic coding," Univ. Delaware, Dep. Elec. Eng., Tech. Rep. 93-2-1,1993.

[5] R. M. Capocelli and A. De Santis, "New bounds on the redundancy of Huffman codes," *IEEE Trans. Inform. Theory*, vol. 37, pp. 1095–1104, July 1991.

[6] R. Elias, "Universal codeword sets and representation of the integers," *IEEE Trans. Inform. Theory*, vol. IT-21, pp. 194–203, Mar. 1975.

[7] S. W. Golomb, "Run-length encodings," *IEEE Trans.Inform. Theory*, vol. IT-12, pp. 399–401, July 1966.

[8] M. Guazzo, "A general minimum-redundancy source-coding algorithm," *IEEE Trans. Inform. Theory*, vol. IT-26, pp. 15–25, 1980.

[9] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proc. IRE*, pp. 1098–1101, Sept. 1952.

[10] R. Hunter and A. H. Robinson, "International digital facsimile coding standards," *Proc. IEEE*, vol. 68, pp. 854–867, July 1980.

[11] M. Jakobsson, "Huffman coding in bit-vector compression." *Inf. Proc. Lett.*, vol. 7, no. 6, pp. 304–307, Oct. 1978.

[12] F. Jelinek and K. Schneider, "On variable-length-to-block coding," *IEEE Trans. Inform. Theory*, vol. IT-18, pp. 765–774, Nov. 1972.

[13] G. G. Langdon, "An introduction to arithmetic coding," *IBM J. Res. Develop.*, vol. 28, pp. 135–149, Mar. 1984.

[14] A. Lempel and J. Ziv, "On the complexity of finite sequences," *IEEE Trans. Inform. Theory*, vol. IT-22, pp. 75–81, Jan. 1976.

[15] N. Merhav and D. L. Neuhoff, "Variable-to-fixed length codes provide better large deviations performance than fixed-to-variable length codes," *IEEE Trans. Inform. Theory*, vol. 38, pp. 135–140, Jan. 1992.

[16] B. L. Montgomery and J. Abrahams, "Synchronization of binary source codes," *IEEE Trans. Inform. Theory*, vol. IT-32, pp. 849–854, Nov. 1986.

[17] ———, "On the redundancy of optimal binary prefix-condition codes for finite and infinite sources," *IEEE Trans. Inform. Theory*, vol. IT-33, pp. 156–160, Jan. 1987.

[18] W. B. Pennebaker, J. L. Mitchell, Jr. G. G. Langdon, and R. B. Arps, "An overview of the basic principles of the *q*-coder adaptive binary arithmetic coder," *IBM J. Res. Develop.*, vol. 32, no. 6 pp. 717–726, Nov. 1988.

[19] J. Rissanen, "Generalized kraft inequality and arithmetic coding."*IBM J. Res. Develop.*, pp. 198–203, May 1976.

[20] J. Rissanen and G. G. Langdon, "Arithmetic coding," *IBM J. Res. Develop.*, vol. 23, no. 2, pp. 149–162, Mar. 1979.

[21] J. Rissanen and K. M. Mohiuddin, "A multiplication-free multialphabet arithmetic code." *IEEE Trans. Commun.*, vol. 37, pp. 93–98, Feb. 1989.

[22] J. Teuhola and T. Raita, "Piecewise arithmetic coding." in *Proc. Data Compression Conf. 1991*, J. A. Storer and J. H. Reif, Eds., Snow Bird, UT, Apr. 1991, pp. 33–42,

[23] B. P. Tunstall, "Synthesis of noiseless compression codes," Ph.D. dissertation, Georgia Inst. Technol., 1967.

[24] T. A. Welch, "A technique for high-performance data compression," *IEEE Comput. Mag.*, pp. 8–19, June 1984.

[25] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression." *Commun. ACM*, vol. 30, pp. 520–540, June 1987.

[26] J. Ziv, "Variable-to-fixed length codes are better than fixed-to-variable length codes for Markov sources," *IEEE Trans. Inform. Theory*, vol. 36, pp. 861–863, July 1990.

[27] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Trans. Inform. Theory*, vol. IT-24, pp. 530–536, Sept. 1978.