Is Huffman Coding Dead?

A. Bookstein[1] and *S.T. Klein*[2]

[1]Center for Information & Language Studies, University of Chicago, Chicago IL 60637, USA
Tel: (312) 702 8268 Fax: (312) 702 0775 Email: bkst@caper.uchicago.edu

[2] Dept. of Mathematics & Computer Science, Bar-Ilan University, Ramat Gan 52900, Israel
Tel: (972–3) 531 8681 Fax: (972–3) 535 3325 Email: tomi@bimacs.cs.biu.ac.il

Abstract: In recent publications about data compression, arithmetic codes are often suggested as the state of the art, rather than the more popular Huffman codes. While it is true that Huffman codes are not optimal in all situations, we show that the advantage of arithmetic codes in compression performance is often negligible. Referring also to other criteria, we conclude that for many applications, Huffman codes should still remain a competitive choice.

1. Introduction

It is paradoxical that, as the technology for storing and transmitting information has gotten cheaper and more effective, interest in data compression has increased. There are many explanations, but most conspicuous is that improvements in media have expanded our sense of what we wish to store. For example, CD-Rom technology allows us to store whole libraries instead of records describing individual items; but the requirements of storing full text easily exceeds the capabilities even of the optical format. Similarly, there is growing interest in storing and transmitting images: in color and at improved resolutions, sometimes in animation. For such cases, data compression can be a very powerful means of increasing the effective capacity of the technology.

Although many *ad hoc* methods have been used over the years, Huffman coding has played a special role as a systematic coding mechanism that has provable optimality characteristics and is easily implemented. However, recent publications about data compression leave one with the impression that Huffman coding has become somewhat out of fashion. These publications stress the suboptimality of Huffman codes, which can be severe in some situations, to the advantage of proposed alternatives. Indeed, the "optimality" of Huffman codes has often been overemphasized in the past and it is not always mentioned that Huffman codes have been shown to be optimal only for *block codes* [1]: codes in which a message is encoded a character at a time, with each new character resulting in a fixed bit pattern being appended to the current code for

the message; this bit pattern is made up of an integral number of bits and is uniquely and instantaneously decodable as the character generating it.

The constraint of the integral number of bits had probably been considered as obvious prior to the development of arithmetic coding, since the possibility of coding elements in fractional bits is quite surprising. Therefore Huffman codes enjoyed widespread popularity in the four decades since their invention.

Arithmetic codes overcome the limitations of block codes. In fact, arithmetic codes have had a long history [1], [37], [38], but became especially popular after Witten, Neal and Cleary's paper [43] in 1987. They are claimed in [43] to be *superior in most respects to the better known Huffman method*, and the general impression one gets from the preponderance of recent papers dealing with or mentioning arithmetic coding is that Huffman coding is outdated, and that ... *everything Huffman codes can do, arithmetic codes can do better* ...

Arithmetic coding has some very strong advantages:

1. they do permit codes that come very close to the entropy bound;
2. they are easily used with an adaptive model, yielding efficient encoding if the alphabet or its characteristics are changing over the file; and
3. the encoding procedure can be naturally and simply extended to encompass even an infinite alphabet. Huffman type codes require special considerations for each such alphabet.

But these advantages come at some cost relative to Huffman codes. Most obvious:

1. arithmetic codes tend to run significantly more slowly than Huffman codes, which can be critical in some applications;
2. they are much less intuitive than Huffman codes and more difficult to explain to a system user;
3. the main advantage of arithmetic codes, their compression effectiveness, does not obtain in most realistic situations. As we shall see below, for text based applications, the savings is typically very small; and
4. in applications in which inaccurate probabilities are used, the savings may actually be negative.

These and other more subtle points discussed below explain Huffman coding's enduring appeal and suggest that it is appropriate at this time to reconsider the value of Huffman coding for compression. We shall try, in this paper, to offer a balanced view of the two approaches, and shall point, in subsequent sections, to various aspects and applications for which Huffman codes should still be the preferred choice. Similar ideas have been mentioned in [3], [25], [34]. We assume the reader is familiar with the details of Huffman codes [26] and arithmetic codes [43], which can be found in most textbooks on compression.

2. How badly do Huffman codes compress?

The compression effectiveness of Huffman codes is sensitive to the characteristics of the alphabet. For purposes of discussion, it is useful to consider separately two extremes in the spectrum of alphabets that one might wish to encode: Large Alphabets and Binary Alphabets.

2.1 Large Alphabet

A database is often composed from an alphabet of a moderate to large number of coding units, for example a natural language alphabet, numbers, and punctuation for a textual database. In this case, a bound by Gallager is often applicable. Gallager [19] showed that the redundancy of a Huffman code is at most $p_1 + 0.086$, where p_1 is the probability of the most frequent codeword. (The *redundancy* is defined as the difference between the average length of a Huffman code and the corresponding entropy, or for our application, by how much Huffman codes are worse than arithmetic codes.) More bounds on the redundancy can be found in Capocelli & al. [8], [9].

For non-pathological cases, the most frequent character of a large alphabet will occur infrequently, and the difference between Huffman and arithmetic coding can become insignificant. Cases in which the most frequent character does occur with high probability may indicate poor alphabet construction, since most characters have little information content. Here, it is often desirable to extend the alphabet to incorporate n-grams that include the frequent characters; in the expanded alphabet, the highest probability will be reduced and Huffman coding could be effective. That Huffman coding is effective for databases of natural text is shown in the first two columns of Table 1, which is discussed in detail below. There we find the Huffman cost (defined as

the relative increase of the compressed file when using Huffman instead of arithmetic coding) is small, but even these values exaggerate the increase since using arithmetic coding requires an explicit end-of-file indication, which incurs an additional storage penalty.

Dealing with End-Of-File. When comparing the effectiveness of arithmetic coding with other techniques, one generally uses the value of the entropy as the measure of effectiveness for arithmetic coding, without taking into account the fact that an additional EOF indication is usually required. This effect is generally small, but a detailed analysis of its impact is useful when trying to assess the often very small compression cost of using Huffman coding.

In general, one stores a large file as several blocks, both to permit more entry points into the file and to limit the propagation of possible errors. Often we don't know the number of characters in a block, as when sentences of natural language text are processed; in this case, an explicit EOF character is needed. But even if a list of the lengths of the blocks is given, or if all the blocks are of the same length, we still lose 1/2 bit per block as an alignment effect: a typical file may in principle require b bits to store, but in practice, $\lceil b \rceil$ bits, since it must be represented in an integral number of bits.

	Huffman	arithmetic	Huffman cost	$K_{\min}$
English	4.1854	4.1603	0.6%	426
Finnish	4.0448	4.0134	0.8%	329
French	4.0003	4.0376	0.9%	269
German	4.1475	4.1129	0.8%	294
Hebrew	4.2851	4.2490	0.8%	279
Italian	4.0000	3.9725	0.7%	383
Portuguese	4.0100	3.9864	0.6%	457
Spanish	4.0469	4.0230	0.6%	451
Russian	4.4704	4.4425	0.6%	377
English–2	7.4446	7.4158	0.4%	363
Hebrew–2	8.0370	8.0085	0.4%	368

Table 1: *Compression of natural language alphabets*

We can now evaluate the increase in the average codeword length, assuming that an EOF element is adjoined after each block. Suppose the file is T characters long, broken up into B possibly variable length blocks, and let $K = T/B$ be the average number of characters per block.

Let $f_1, \ldots, f_n$ be the frequencies of the n elements of the alphabet, $\{a_i : i = 1 \cdots n\}$, in the given text, so $T = \sum_{i=1}^{n} f_i$. The corresponding probabilities are $p_i = f_i/T$ and the entropy is $H = -\sum_{i=1}^{n} p_i \log_2 p_i$. If we add one EOF element, a_0, to the end of each block, we are actually encoding $T' = T + B$ elements, and the new set of probabilities is $\{p'_i\}_{i=0}^{n}$, with $p'_0 = B/(B+T)$ and $p'_i = f_i/(T+B) = p_i T/(T+B)$, for $0 < i \leq n$. The length of the compressed file will now be

$$S \;=\; \frac{B}{2} - (B+T) \sum_{i=0}^{n} p'_i \log_2 p'_i,$$

where the first term indicates the alignment loss of 1/2 bit per block. We can now amortize the total length over the T original elements in the file, to get the increase, $\Delta(K)$, in average codeword length:

$$\begin{aligned} \Delta(K) &= \frac{1}{T} S - H. \\ &= \frac{1}{2K} + \left(1 + \frac{1}{K}\right) \log_2 (K+1) - \log_2 K. \end{aligned}$$

Note that $\Delta(K)$ depends only on the block size K, and not on the probability distribution at hand. Considering $\Delta(K)$ as a continuous function, we find that the derivative $\frac{\partial \Delta}{\partial K}$ is negative for all K, thus $\Delta(K)$ is strictly decreasing, as expected, and $\lim_{K \to \infty} \Delta(K) = 0$.

Data. It is not easy to agree on what a "typical" probability distribution is. We therefore decided to run our tests of relative compression performance on the character distributions of various natural languages. Even though more sophisticated models exist, the simple encoding of the individual characters or character pairs is often preferred. The distribution of the 26 letters and the 371 letter pairs of English was taken from Heaps [22]; the distribution of the 29 letters of Finnish is from Pesonen [36]; the distribution for French (26 letters) has been computed from the database of the *Trésor de la Langue Française* (TLF) of about 112 million words (for details on TLF, see [6]); for German, the distribution of 30 letters (including blank and *Umlaute*) is given in

Bauer & Goos [2]; for Hebrew (30 letters including two kinds of apostrophes and blank, and 735 bigrams), the distribution has been computed using the database of The Responsa Retrieval Project (RRP) [14] of about 40 million Hebrew and Aramaic words; the distribution for Italian, Portuguese and Spanish (26 letters each) can be found in Gaines [18], and for Russian (32 letters) in Herdan [23]. The results are summarized in Table 1, the two last lines corresponding to the bigrams.

The first two colums list the average codeword length of Huffman codes and arithmetic codes respectively, and the third column gives the increase of the former over the latter in percent. We see that this increase is very low, at most one percent, without taking the EOF character into account. The fourth column lists for each language the minimal average block size, $K_{\min}$, for which arithmetic codes improves compression, that is

$$K_{\min} = \min\{K \mid \text{average for arithmetic codes} + \Delta(K) < \text{average for Huffman codes}\}$$

(the block size being measured in number of encoded elements). In other words, only if the average blocksize is chosen larger or equal to $K_{\min}$ does it pay to use arithmetic codes. For example, for English, if the average blocksize is smaller than 426 elements, Huffman codes would yield better compression. The surprising result here is that these $K_{\min}$ values are relatively large. If, for example, each block consisted of a single sentence (a reasonable assumption in an information retrieval system), given that the average length of a word (including the following blank) in English is 5.38 characters [22], arithmetic codes give better compression only for sentences of at least 79 words! Very few sentences are that long. Moreover, even if the blocksize is larger than $K_{\min}$, the EOF effect still reduces the Huffman cost, shown in column 3. For example, choosing $K = 1000$ lowers the values in the third column by between 0.2 and 0.3.

We conclude that even though there is, in principle, a compression advantage for arithmetic over Huffman codes, this advantage is so small that it might often be negligible, and if the additional overhead of arithmetic codes is taken into account, the advantage may vanish completely. These conclusions are consistent with the extensive empirical tests carried out by Moffat & al. [33].

2.2 Small Alphabet

The possibility of poor compression efficiency for Huffman codes is, in practice,

most pronounced for small alphabets. Consider for example a binary alphabet, with the two letters appearing with probabilities ε and $1-\varepsilon$ respectively. The length of a codeword for any binary alphabet is exactly 1 bit with Huffman coding. However, with arithmetic coding, the entropy $-\varepsilon \log_2 \varepsilon - (1-\varepsilon)\log_2(1-\varepsilon)$ could be reached; in this case, the ratio of the size of the Huffman encoded file to the arithmetically encoded file would tend to infinity as $\varepsilon \to 0$. In such a situation, the argument for using arithmetic encoding is most pursuasive.

But this is an extreme case. Clearly, if, in a real application, the probabilities are close to .5, then there is little advantage in using any type of compression method. But even if the probabilities are very skewed, it is often possible to reformulate the problem in such a way that Huffman coding is acceptable. As in the large alphabet case, a skewed probability distribution may well indicate alphabet mis-specification, and suggests that we redefine the alphabet. The classical approach is to use blocking: instead of treating individual bits as members of the alphabet, use blocks of bits. This extends the alphabet and creates a probability distribution that is not so skew. Because of the Gallager bound, for the new alphabet, Huffman codes can be quite good. Moffat & al. [33] also found that the case of arithmetic coding is strongest for highly skewed (although artificially obtained) alphabets, but did not try minor alphabet redefinitions.

But the possibility of blocking is only one example in which a highly skewed probability distribution may be an indicator that the wrong alphabet is being used. For example, if one of the characters of the binary alphabet is very unlikely, it suggests that *run length* encoding is more appropriate. To illustrate this point, consider the compression of large, sparse bitmaps. Such bitmaps often serve as occurrence maps for the different terms of a large full-text information retrieval system, where they speed up the processing of Boolean queries. All the maps are of the same length which equals the total number of documents in the system, and there is one map for each term (or for terms appearing more often than some threshold); bit i of map j is 1 if and only if term j appears in document number i. The advantage of such bitmaps is that Boolean queries on keywords are easily translated to corresponding logical operations on bitmaps, speeding up the retrieval process. The disadvantage of the maps is their size (several hundreds of MB on certain systems), making them unpractical, unless compression methods are used to shrink them considerably.

	n	total millions	Huffman MB	arithmetic MB	increase %
each bit	2	650.1	77.49	9.64	703
k-blocks	256	81.3	13.70	7.70	78
POW2	266	14.2	8.26	8.22	0.5
LLRUN	267	9.3	7.354	7.316	0.5

Table 2: *Compression of sparse bitmaps*

Table 2 compares the compression effectiveness for various expansions of a binary alphabet. It is based on a file of 15378 bitmaps taken from RRP; each map is 5284 bytes in length, representing 42272 documents. The first column describes the alphabet used. The column entitled n gives the alphabet size; the next column gives the total number of occurrences in millions; the next two columns give the size of the file in Megabytes, if compressed by Huffman or arithmetic coding; the last column shows by how much the Huffman encoded file is larger than the arithmetically encoded one.

As a binary file, one could encode the 0 and 1 bits individually. This obviously gives no compression by Huffman coding, but could be quite efficient with arithmetic codes. The consequence of using this alphabet is shown in row 1. We then increase the size of the alphabet in order to improve the compression effectivenes of Huffman codes (and in fact also for arithmetic codes). Simplest is to encode all blocks of k consecutive bits [27]. The second line of Table 2, corresponding to this method for $k = 8$, shows there is still a clear advantage of using arithmetic codes, though the benefit has been considerably reduced. The problem is that, even with 256 elements to be encoded, the distribution is still very skew. The overall frequency of 1-bits in the file is only about 1.7% and a block of 8 consecutive zeros has probability 0.925 (it is larger than $(1 - 0.017)^8$ because the 1-bits are not uniformly scattered through the maps). The next step consists therefore of generating also codewords for *runs* of 0-blocks of various lengths.

Several such methods are suggested in [15]. The results of methods POW2 and LLRUN of [15] appear in the third and fourth lines of Table 2; the benefit of arithmetic codes has now been reduced to merely half a percent.

Noting the effect of the EOF requirement of arithmetic codes further reduces the

Huffman cost. In our case, each bitmap has to be accessible individually. Adding thus an EOF to each of the 15378 maps, and half a bit on the average for bit alignment, this would, for the last two lines of Table 2, raise the size of the arithmetically compressed file to 8.24MB and 7.34MB respectively, reducing the Huffman cost (last column) to only 0.2% for both lines. Thus this small effect cuts in half the benefit of arithmetic encoding.

3. The case of inaccurate probabilities

Arithmetic codes give better compression than Huffman codes if, as one generally assumes, the probabilities are given and correct. Regretfully, this is not always the case, especially when the probabilities on which the code is based are derived from a mathematical model. The problem of inaccurate source probabilities has been considered by Gilbert [20], who suggests that we avoid the inefficiency caused by underestimating a probability (which leads to assigning longer codewords than needed) by bounding the depth of the Huffman tree. An optimal linear procedure for bounding the depth can be found in [31].

We saw above that Huffman codes may theoretically be infinitly worse than arithmetic codes for certain probability distributions; but if our estimates are wrong, just the opposite may be true. Consider the set of probabilities $\{\frac{1}{3}+\varepsilon, \frac{1}{3}, \frac{1}{3}-2\varepsilon, \varepsilon\}$, for $0 < \varepsilon \leq \frac{1}{6}$; the corresponding Huffman codewords have lengths $\{1, 2, 3, 3\}$ respectively, so the average codeword length is $2-2\varepsilon$. For $\varepsilon = 10^{-3}$, one gets an average of 1.998 for Huffman codes and 1.594 for arithmetic codes. But suppose the above estimates were wrong and the true distribution was in fact uniform, i.e., $\{\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}\}$. The previous Huffman tree is no longer optimal, and the actual average codeword length using codes based on the incorrect Huffman tree is $A_h = 2.25$. For arithmetic codes on the other hand, if we use codes based on the incorrect probabilities, the actual average codeword length, A_a, tends to $\frac{3}{4}\log_2(3) - \frac{1}{4}\log_2 \varepsilon$ as $\varepsilon \to 0$. For example, for $\varepsilon = 10^{-3}$, $A_a = 3.68$, which is already larger than the corresponding Huffman average, and as $\varepsilon \to 0$, we get $A_a/A_h \to \infty$.

The use of incorrect probabilities is not very unusual. In an application to model-based bitmap compression [5], Huffman codes gave consistently better compression than arithmetic codes. This was explained by the fact that the model predicted many very low probabilities for certain bit patterns, but the very fact that some of these bit

patterns actually appeared, showed that their probabilities had been underestimated.

Such wrong guesses may indeed lead to very inefficient codes. Ernest Wright wrote in 1939 a novel called *Gadsby*, in which the letter `E` never appears. If one uses this novel to estimate the character frequencies in English, the Huffman codeword assigned to `E` would be 14 bits long (based on the first 10028 words of the novel, and assuming a frequency of 1 for `E`), instead of just 3 bits on regular English text. For arithmetic codes, each `E` would add 15.4 bits. Taking the full distribution into account, basing ourselves on Gadsby's frequencies but encoding regular English text, the average Huffman codeword length would increase from 4.19 to 5.46, and for arithmetic codes from 4.16 to 5.60! We thus see that arithmetic codes give worse compression in this case, even without considering the overhead caused by EOF.

Table 3 summarizes an experiment in which we took the probability distributions of English, German, Finnish and French (as in Table 1), adding to them the character distribution in Gadsby, and checked what happens if they are mutually interchanged. The rows correspond to the distributions which are used to generate the codewords (the **assumed** distribution), and the columns correspond to the distribution that actually occurs (the **true** distribution). For each pair of distributions (A, B), both Huffman and arithmetic codes were computed, and the table lists at the intersection of row A with column B by how much (in percent) the average Huffman codeword is longer than the average arithmetic codeword, i.e., the value $(A_h/A_a - 1) \times 100$. A negative value thus means that for the given pair, Huffman codes do better than arithmetic codes.

	English	**Gadsby**	**German**	**Finnish**	**French**
English	0.6	1.4	-2.1	-5.1	0.4
Gadsby	-2.4	1.0	-2.9	-0.8	-3.3
German	-1.8	-3.6	0.9	-5.2	-0.1
Finnish	2.4	1.2	2.7	0.8	2.9
French	0.8	2.2	-4.6	-11.3	0.9

Table 3: *Excess of Huffman over arithmetic in percent when assuming the rows to compress the columns*

For example, if we assume a character distribution like in a German text, but use this to encode an English text, the average arithmetic codeword length would be

1.8% longer than the average Huffman codeword length. For computing the table, we assumed a 32 character alphabet for each of the 5 distributions, since there are 3 letters in German and 3 others in Finnish which do not exist in other languages. We used a frequency of 1 for each non-occurring letter. Huffman codes could have dealt easily also with probabilities that are zero, but for arithmetic codes, a probability of 0 causes problems.

An interesting point about Table 3 is the fact that so many entries are negative. The diagonal corresponds to assuming the true distribution, so clearly all the values there must be positive. But of the remaining 20 entries, 12 have negative values. Moreover, in absolute value, the negative entries seem larger than the positive ones: the average of the positive entries is 1.4 while the average of the negative entries is -3.6. We therefore conclude that if the probabilities are not sure to be known accurately, or if they are estimated by means of a model which could generate very low values, we might be better off by using Huffman codes rather than arithmetic codes.

But even if the codes are based on actual statistics, errors could appear. It is not uncommon, in a dynamic file, to create a code with a portion of the file and then continue to use that code as new records arrive. Even if the records are generated by the same mechanism as the earlier records, statistical fluctuations occur [7]. But, more seriously, the record generation mechanism is rarely static, and considerable probability drift could occur. In such a situation, the deterioration in compression might be serious.

4. Time considerations

One of the major advantages of Huffman codes is its speed. There is general agreement that Huffman codes are faster than arithmetic codes, though the reported degree of improvement varies from *a bit faster for encoding and decoding* [43] to up to *40 times faster for decoding* [34] (although it is shown in [33] that this issue must be reconsidered when adaptive coding is used).

The construction of a Huffman code can be done in $O(n \log n)$, where n is the size of the alphabet [41]; relative to the size of the text, this is a constant. Once the code is given, encoding consists of concatenating fixed bit-strings. Decoding is slightly more involved, as a binary tree is traversed, guided by the sequence of bits of the compressed file. On the other hand, arithmetic codes require multiplications

and sometimes also divisions [25], which clearly are more time consuming; however the extent of performance deterioration depends on many implementation details.

In the following experiment, we used the routines supplied with [35], compiled by the Turbo C++ compiler, on a 16 MHz 386SX machine. We chose the text files `Gadsby` (as above) and the Hebrew `Pentateuch`; to test also non-text files, we also used `gnuplot.exe`, the executable file of the Gnuplot V.2 Shareware program, and `chess.bmp`, a picture bitmap supplied as part of the Windows 3.0 system. Table 4 lists the encoding and decoding times in seconds.

	size	**encoding**			**decoding**		
	K	arith	Huff	adap H	arith	Huff	adap H
`Gadsby`	56	23.1	10.4	25.0	90.1	13.8	24.5
`Pentateuch`	437	182.9	84.6	190.6	775.7	108.4	184.8
`gnuplot.exe`	204	109.9	53.7	143.3	339.5	74.1	139.6
`chess.bmp`	150	42.2	16.6	40.0	228.8	22.1	38.1

Table 4: *Comparison of processing speed*

Note that encoding for arithmetic codes took more than twice as long as for Huffman codes, and decoding up to 10 times as long! Since arithmetic codes can easily be used with an adaptive model, it is perhaps more fair to compare them with adaptive Huffman codes [19], [42], [30], as done in [43]. Our results (columns headed 'adap H') were however different from those reported in [43], yielding a decoding speed up to 6 times faster for adaptive Huffman codes than for arithmetic codes.

There have been attempts to improve the speed of arithmetic codes, either approximating the multiplication operations [10], or approximating the probabilities [25], or using a mixture of block and nonblock coding [40]. These methods, however, gain in speed by sacrificing the compression optimality.

The slow decoding speed of arithmetic codes can be a significant disadvantage, because in many compression applications, encoding and decoding are not symmetrical tasks. For instance, in large static information retrieval systems [28], [3], encoding is done only once when the system is set up, but decoding is needed each time the

compressed files are accessed. For such applications, encoding time is not critical, but decoding must be very fast.

Several methods are presented in [11] that allow accelerated decoding of a Huffman encoded file, using a set of m *partial decoding tables*. The coded input is processed on a block-per-block basis, rather than bit-per-bit, where each block consists of k bits, and k is chosen to facilitate computer manipulation (e.g., $k = 8$, yielding a byte oriented routine). The number of entries in each table is 2^k, corresponding to the 2^k possible values of the k-bit patterns. Each entry is of the form (W, ℓ), where W is a sequence of characters and ℓ $(0 \leq \ell < m)$ is the index of the next table to be used. The idea is that entry i, $0 \leq i < 2^k$, of table number 0 contains, first, the longest possible decoded sequence W of characters from the k-bit block representing the integer i (W may be empty when there are codewords of more than k bits); usually some of the last bits of the block will not be decipherable, being the prefix P_ℓ of more than one codeword; ℓ will then be the index of the table corresponding to that prefix (if P_ℓ = the empty string, then $\ell = 0$). Table number ℓ is constructed in a similar way except for the fact that entry i will contain the analysis of the bit pattern formed by the prefixing of P_ℓ to the binary representation of i. We thus need a table for every possible proper prefix of the given Huffman code, so that $m = n - 1$, where n is the size of the alphabet.

Suppose, for example, that our alphabet is {`A, B, C, D`}, and that the corresponding Huffman codewords are $\{0, 11, 100, 101\}$. There are thus 3 possible proper prefixes: the empty string, 1 and 10, so that 3 tables are needed. If we choose $k = 3$, each table will have 8 entries. The tables are depicted in Figure 1.

Entry	Pattern for Table 0	Table 0		Table 1		Table 2	
		W	ℓ	W	ℓ	W	ℓ
0	000	`AAA`	0	`CA`	0	`CAA`	0
1	001	`AA`	1	`C`	1	`CA`	1
2	010	`A`	2	`DA`	0	`C`	2
3	011	`AB`	0	`D`	1	`CB`	0
4	100	`C`	0	`BAA`	0	`DAA`	0
5	101	`D`	0	`BA`	1	`DA`	1
6	110	`BA`	0	`B`	2	`D`	2
7	111	`B`	1	`BB`	0	`DB`	0

Figure 1: *Partial decoding tables*

The column headed 'Pattern' contains for every entry the binary string which is decoded in Table 0; the binary strings which are decoded by Tables 1 and 2 are obtained by prefixing '1', resp. '10', to the strings in 'Pattern'. For example, entry number 5 of table 2 contains the decoding of the binary string 10101, which yields `DA` and we are left with a remainder of 1; the next table to be accessed will thus be the table corresponding to this proper prefix, which is table 1. The decoding procedure is thus extremely simple and fast: $M(i)$ denotes the i-th block of the input stream, ℓ is the index of the currently used table and $T(\ell, j)$ is the j-th entry of table ℓ:

$\ell \leftarrow 0$
for $i \leftarrow 1$ **to** length of input **do**
    $(\text{output}, \ell) \leftarrow T(\ell, M(i))$
end

The storage requirements for these tables are generally reasonable (about 25K for a 26 characters English alphabet distribution), but if the alphabet is larger or RAM is scarce, the necessary space can be reduced, see [11].

This decoding method is not restricted to Huffman codes, but can be applied to any method for which every member of the alphabet is always encoded in the same way. This is not true for adaptive methods. For arithmetic codes, it is not true even for its static variant. Having each character always encoded in the same way may also be used to improve certain search problems. Suppose we are given a large file X in which we wish to locate a substring Y, but assume that X is stored in its compressed form $\mathcal{C}(X)$. The obvious way to procede is to decompress $\mathcal{C}(X)$ and to search for Y in $\mathcal{D}(\mathcal{C}(X)) = X$. But in our case, we could instead encode the pattern, which is in general much shorter than the text, and search for $\mathcal{C}(Y)$ in $\mathcal{C}(X)$. The search might be trickier now, but will often yield significant savings.

5. Communicating the Code

To decode a compressed file, the decoder must know the code that was used. Adaptive methods do this by keeping the encoder and decoder in synchonization as the file is encoded. These codes often yield good compression, but can be slow, since the model needs constant updating. But if an invariant code is used, it must be transmitted together with the encoded file.

Coding the code is not usually considered a problem, since the size of the description of the code depends only on the alphabet, and is thus generally independent of the size of the text itself. But for smaller files, this overhead is not always negligible, and for more sophisticated models, like basing compression on a first order Markov chain of an extended alphabet [4], the necessary header may be of considerable size.

Most implementations of arithmetic codes use 2 bytes to store the cumulative probabilities, yielding a precision of 2^{-14}. This might be enough for many applications, but not for all. If a large text is encoded based on word frequency counts [3], higher precision may be needed, usually 4 bytes for each probability. For Huffman codes, on the other hand, the frequencies need not be transmitted, nor the codewords themselves. In fact, it suffices for both encoder and decoder to know the *lengths* of the codewords, as both could construct, based on those lengths, the same optimal code. A natural choice would be a *canonical* code [39], [21], [16]. An easy way to generate such a code, which is needed in the encoding phase, is as follows [21]: given are the lengths $\ell_1, \ldots, \ell_n$ of the Huffman codewords in non-decreasing order (thus corresponding to the probabilities that have been sorted into non-increasing order), the i-th codeword consists of the ℓ_i first bits to the right of the "binary point" in the binary representation of $\sum_{j=1}^{i-1} 2^{-\ell_j}$.

For decoding, the corresponding Huffman tree is needed, which can be constructed by the following simple procedure [16] in linear time. The idea is to pass sequentially over the vector of lengths $\{\ell_i\}$ and to simulate a depth first traversal of a binary tree which is built by the procedure itself; i.e., when passing to a left or right son which has not yet been defined, a new node is allocated and linked into the tree. During this traversal, every time a level is reached which equals the current value of ℓ_i, the procedure passes to ℓ_{i+1} and considers the current node v as a leaf (thus the next node to be visited will be the father of v).

The string of codeword lengths $\{\ell_1, \ldots, \ell_n\}$ can be represented compactly as $\langle n_1, \ldots, n_k \rangle$, where n_i is the number of codewords of length i, and $k = \ell_n$ is the maximal length (or depth of the Huffman tree). Such a string is called a *quantized source* in [13], and satisfies $\sum_{i=1}^{k} n_i 2^{-i} = 1$ [29, Exercise 2.3.4.5–3]. The quantized source is all we need to construct the Huffman code, but in order to assign the proper codewords to the corresponding characters, one also needs the list of characters sorted by frequency. There are several ways of doing this.

Method **A**. A direct way to encode the code is by the sequence $k, n_1, \ldots, n_k, a_1, a_2, \ldots, a_n$, where $n = \sum_{i=1}^{k} n_i$ and $a_1, \ldots, a_n$ is the sorted list of characters. For simplicity, we can assume that k and each n_i is encoded in one byte, though a *universal* encoding of the integers can be used [12], encoding the integer x in $O(\log x)$ bits.

Method **B**. If n is large, we might be better off not sorting the characters, but simply listing the lengths of the codewords in the alphabet's natural order. Characters that don't appear are indicated by length 0. The lengths of the codewords can mostly be coded in half a byte each — they rarely exceed 16 bits.

Method **C**. If enough characters don't appear, it might be most efficient to indicate the characters that do appear by a bitmap. For example, many applications assume that the basic alphabet is the set of the 256 8-bit patterns. Instead of the full list of characters, a 32-byte bit-vector can be used to indicate which characters appear in the given text.

Method **D**. A final method can be used when the set of characters appearing in a given text consists of several runs of consecutive elements from a well-known basic set like ASCII (e.g., upper case, lower case, digits, etc.) In this case, it could be more economical to encode the given alphabet by a list of pairs, indicating the beginning and end of each run, preceded by the number of runs; e.g., (3, (LF, LF), (A, Z), (0, 9)) for an alphabet consisting only of line-feed, upper case letters and digits.

Method **A** is specific to Huffman codes, since it relies on the fact that the possible lengths of the codewords are integers. The other methods are variants for the encoding of the character set: omitting it by assuming the natural order (**B**), using a bitmap (**C**) or lists (**D**). The chosen representation of the character set can be used for both Huffman and arithmetic codes. The difference between the two would then be in the number of bits needed to encode the lengths of the codewords for Huffman codes, or the probabilities for arithmetic codes. Since the lengths generally fit into 4 bits, but at least 2 bytes are needed for the probabilities, we may count an excess of 1.5 bytes per element of the alphabet for using arithmetic codes. For more details on coding the code, the reader is referred to [24].

Take for example the distribution of the characters in English as given in [22]. The corresponding quantized source is $\langle 0, 0, 2, 7, 7, 5, 1, 1, 1, 2 \rangle$; Method **A** would precede this sequence by $k = 10$, the depth of the tree, and follow it by the 26 characters in order of

frequency: E, T, A, O, I, N, S, R, H, L, D, U, C, F, M, W, Y, G, P, B, V, K, X, J, Q, Z. Storing k, each n_i and each character in one byte, we would thus need 37 bytes here. To store the alphabet with methods **B**, **C** and **D** one needs 26, 32 and 3 bytes respectively, to which 13 bytes have to be added for the lengths of the Huffman codewords, or 52 bytes for the probabilities for arithmetic codes. From Table 1 we know that the average loss per character by using Huffman instead of arithmetic codes is 0.0251 bits, so the text has to be at least of length 12431 characters to justify the excess of these 39 bytes for methods **B**, **C** and **D**. For a larger alphabet, the difference is even more striking. Referring to the 371 English bigrams, the excess of arithmetic over Huffman codes for the last three methods would be of 556.5 bytes, justified only by a text of at least 309167 characters. Methods **C** and **D** might be less efficient for bigrams, but if all character pairs have to be listed, one can list them in non-increasing order of frequency, so that Method **A** could be used with Huffman codes, adding to the list of character pairs just the 14 bytes to represent the quantized source $\langle 0, 0, 0, 0, 3, 14, 31, 59, 58, 70, 32, 40, 64\rangle$. Thus the excess would be of 728 bytes, which are amortized only by texts of length exceeding 404444 bytes! If the coding is based on a Markov chain, i.e., there is a different code for the set of successors of each character [4], many codes must be represented, so that the difference between Huffman codes and arithmetic codes might be considerable. We again conclude that the compression gain of arithmetic codes might easily be lost in most applications.

6. Robustness against errors

In many situations, especially when an encoding method is chosen to cut down transmission costs in a communication system, an important criterion is the ability of the code to recover from minor errors. It is well known that static Huffman codes generally tend to resynchronize quickly after a transmission error [32]. There are of course exceptions: if all the codewords are of even length and a bit is lost or an extraneous bit is picked up, synchronization is lost forever.

More generally, suppose an error has occurred and that x is the last codeword, following the location of the error, that is not correctly decoded before synchronization is regained; then there exists a codeword y such that either x is a proper suffix of y or vice versa. It follows that if a Huffman code has the *affix* property [17], i.e., no codeword is the prefix or the suffix of any other codeword, it will be never-self-synchronizing after

an error [21]. But such affix codes are extremely rare. There are more than 120 million different Huffman codes for the quantized source of English $\langle 0, 0, 2, 7, 7, 5, 1, 1, 1, 2\rangle$, none of which has the affix property. In fact, the only quantized source for $n = 26$, with up to 11 levels, for which an affix code exists, is $\langle 0, 1, 1, 3, 9, 8, 4\rangle$ [17].

The following experiment should illustrate the ability of Huffman codes to recover after errors. Using again the routines supplied in [35] to encode the beginning of `Gadsby`, we complemented a single bit in the encoded file, and tried to decode the resulting string. Line (n) below, $n = 0, 1, \ldots, 5$, is the decoding of the file in which bit number $80n + 1$ has been complemented. For clarity, blanks are replaced by dashes, and garbeled characters appear in bold-face.

(0) **-dniaao**outh,-throughout-all-history,-had-had-a-champion-to-stand-up-for-it;- ...

(1) If-Youth,-throu**m**out-all-history,-had-had-a-champion-to-stand-up-for-it;- ...

(2) If-Youth,-throughout-all-history,-**i-c–dhl-l**champion-to-stand-up-for-it;-to-show- ...

(3) If-Youth,-throughout-all-history,-had-had-a-champion-**--ovniwlio-y**for-it;-to-show- ...

(4) ...a-champion-to-stand-up-for-it**rl**to-show-a-doubting-world-that-a-child-can-think;- ...

(5) ...a-champion-to-stand-up-for-it;-to-show-a-doubtin**hprcyh–wM**-a-child-can-think;- ...

As can be seen, synchronization is always regained after only a few wrong characters. For arithmetic codes, on the other hand, the characters are not encoded individually, but the whole message is represented by a single real number. Changing a bit means changing the number, which generally will produce a completely different message. Repeating the experiment above with the arithmetic encoding of the beginning of `Gadsby`, synchronization was never regained, for all values of n tested. As example, the following string has been produced by decoding the file in which bit number 161 was complemented:

If-Youth,-throughout-all-history,-**ghaiolsne;-a-iaicatfmi-aU-wml-b-hootdltr-ko** ...

If adaptive Huffman codes are to be used, we are more vulnerable to errors. It might well be that the decoder resynchronizes after an error, but unlike with the static case, this does not mean that the tail will be correctly decoded. Adaptive codes rely on the assumption that encoder and decoder gradually build identical codes. If some characters have been garbeled, the decoder's Huffman tree might be different enough to trigger more errors, changing the tree even further, etc. But adaptive Huffman codes may give considerable savings in certain applications, sometimes even justifying

the use of error-correcting codes. Using such codes is rarely appropriate in connection with arithmetic codes, whose primary justification is precisely the savings lost by using an error correcting code.

7. Conclusion

For many compression projects, a clearcut division between model building and coding is possible, with Huffman codes traditionally dominating the field with regard to the coding phase. More recently, arithmetic coding has risen as a serious alternative — indeed, for a period of time it seemed that Huffman codes would be supplanted by the newer technique. We now have had considerable practical and theoretical experience with both methods, and a judicious statement of the relative advantages of both is possible. This was the objective of this paper, resulting in the conclusion that for a substantial portion of compression applications, Huffman coding, because of its speed, simplicity and effectiveness, is likely to be the preferred choice. On the other hand, for adaptive coding, or when dealing with highly skewed alphabets that cannot be redefined, arithmetic coding may well be the better of the two.

However, when adaptive coding is appropriate, a comparison limited to Huffman codes and arithmetic codes may not be adequate. Our comments are intended for applications, such as the preparation of a CD-Rom, in which preliminary statistical analyses and code definition are possible. For other situations, another large class of widespread compression techniques: those derived from the work of Ziv and Lempel, become attractive. Their two algorithms LZ77 [44] and LZ78 [45] are the basis of many popular methods, such as the Unix *compress* command, PKZIP, ARJ, and many others. They are fast and often achieve better compression than simple static Huffman coding. But they have a somewhat different field of application, and do not compare as directly to Huffman codes as arithmetic codes do. We therefore leave a comparison of Huffman codes with Lempel-Ziv type encoding to a subsequent paper.

References

[1] **Abramson N.,** *Information Theory and Coding*, McGraw-Hill, New York (1965).

[2] **Bauer F.L., Goos G.,** *Informatik, Eine einführende Übersicht, Erster Teil,* Springer Verlag, Berlin (1973).

[3] **Bell T.C., Moffat A., Nevill C.G., Witten I.H., Zobel J.,** Data compression in full text retrieval systems, to appear in *J. ASIS.*

[4] **Bookstein A., Klein S.T.,** Compression, Information Theory and Grammars: A Unified Approach, *ACM Trans. on Information Systems* **8** (1990) 27–49.

[5] **Bookstein A., Klein S.T.,** Models of Bitmap Generation: A Systematic Approach to Bitmap Compression, *Information Processing & Management* **28** (1992) 735–748.

[6] **Bookstein A., Klein S.T., Ziff D.A.,** A systematic approach to compressing a full text retrieval system, *Information Processing & Management* **28** (1992) 795–806.

[7] **Bookstein A., Klein S.T., Raita T., Ravichandra Rao I.K., Patil M.D.,** Can random fluctuations be exploited in data compression, *Proc. DCC'93* (1993).

[8] **Capocelli R.M., Giancarlo R., Taneja I.J.,** Bounds on the redundancy of Huffman codes, *IEEE Trans. on Inf. Th.,* **IT–32** (1986) 854–857.

[9] **Capocelli R.M., DeSantis A.,** New bounds on the redundancy of Huffman codes, *IEEE Trans. on Inf. Th.,* **IT–37** (1991) 1095–1104.

[10] **Chevion D., Karnin E.D., Walach A.C.,** High efficiency, multiplication free approximation of arithmetic coding, *Proc. DCC'91* (1991) 43–52.

[11] **Choueka Y., Klein S.T., Perl Y.**, Efficient variants of Huffman codes in high level languages, *Proc. 8-th ACM-SIGIR Conf.,* Montreal (1985) 122–130.

[12] **Elias P.,** Universal codeword sets and representation of the integers, *IEEE Trans. on Inf. Th.,* **IT–12** (1975) 194–203.

[13] **Ferguson T. J., Rabinowitz J. H.**, Self-synchronizing Huffman codes, *IEEE Trans. on Inf. Th.* **IT–30** (1984) 687–693.

[14] **Fraenkel A.S.,** All about the Responsa Retrieval Project you always wanted to know but were afraid to ask, Expanded Summary, *Jurimetrics J.* **16** (1976) 149–156.

[15] **Fraenkel A.S., Klein S.T.**, Novel compression of sparse bit-strings, *Combinatorial Algorithms on Words*, NATO ASI Series Vol **F12**, Springer Verlag, Berlin (1985) 169–183.

[16] **Fraenkel A.S., Klein S.T.,** Bounding the depth of search trees, to appear in *The Computer Journal* (1993).

[17] **Fraenkel A.S., Klein S.T.,** Bidirectional Huffman coding, *The Computer Journal* **33** (1990) 296–307.

[18] **Gaines H.F.,** *Cryptanalysis, A Study of Ciphers and their solution*, Dover Publ. Inc., New York (1956).

[19] **Gallager R.G.,** Variations on a theme by Huffman, *IEEE Trans. on Inf. Th.,* **IT–24** (1978) 668–674.

[20] **Gilbert E.N.,** Codes based on inaccurate source probabilities, *IEEE Trans. on Inf. Th.* **IT–17** (1971) 304–314.

[21] **Gilbert E.N., Moore E.F.**, Variable-length binary encodings, *The Bell System Technical Journal* **38** (1959) 933–968.

[22] **Heaps H.S.,** *Information Retrieval, Computational and Theoretical Aspects*, Academic Press, New York (1978).

[23] **Herdan G.,** *The Advanced Theory of Language as Choice and Chance*, Springer-Verlag, New York (1966).

[24] **Hirschberg D.S., Lelewer, D.A.,** Efficient decoding of prefix codes, *Comm. ACM* **33** (1990) 449–459.

[25] **Howard P.G., Vitter J.S.,** Practical implementations of arithmetic coding, Tech. Rep. CS-92–18, Dept. of CS, Brown University (1992).

[26] **Huffman D.,** A method for the construction of minimum redundancy codes, *Proc. of the IRE* **40** (1952) 1098–1101.

[27] **Jakobsson M.**, Huffman coding in bit-vector compression, *Inf. Proc. Letters* **7** (1978) 304–307.

[28] **Klein S.T., Bookstein A., Deerwester S.,** Storing Text Retrieval Systems on CD-ROM: Compression and Encryption Considerations, *ACM Trans. on Information Systems* **7** (1989) 230–245.

[29] **Knuth D.E.,** *The Art of Computer Programming, Vol* **I***, Fundamental Algorithms*, Addison-Wesley, Reading, Mass. (1973).

[30] **Knuth D.E.,** Dynamic Huffman coding, *J. of Algorithms* **6** (1985) 163–180.

[31] **Larmore L.L., Hirschberg D.S.,** A fast algorithm for optimal length limited Huffman codes, *Journal ACM* **37** (1990) 464–473.

[32] **Lelewer D.A., Hirschberg D.S.,** Data compression, *ACM Computing Surveys* **19** (1987) 261–296.

[33] **Moffat A., Sharman N., Witten I.H., Bell T.C.,** An Empirical evaluation of coding methods for multi-symbol alphabets, *Proc. DCC'93* (1993).

[34] **Moffat A., Zobel J.,** Coding for compression in full-text retrieval systems, *Proc. DCC'92* (1992) 72–81.

[35] **Nelson M.,** *The Data Compression Book*, M & T Publishing, Inc., (1991).

[36] **Pesonen J.,** Word inflexions and their letter and syllable structure in Finnish newspaper text, Research Rep. 6/1971, Dept. of Special Education, University of Jyräskylä, Finland (in Finnish, with English summary).

[37] **Rissanen J.J.,** Generalized Kraft inequality and arithmetic coding, *IBM J. Res. Dev.* **20** (1976) 198–203.

[38] **Rissanen J.J., Langdon G.G.,** Arithmetic coding, *IBM J. Res. Dev.* **23** (1979) 149–162.

[39] **Schwartz E.S., Kallik B.,** Generating a canonical prefix encoding, *Comm. ACM* **7** (1964) 166–169.

[40] **Teuhola J., Raita T.,** Piecewise arithmetic coding, *Proc. DCC'91* (1991) 33–42.

[41] **Van Leeuwen J.,** On the construction of Huffman trees, *Proc.* 3^{rd} *ICALP Conference*, Edinburgh University Press (1976) 382–410.

[42] **Vitter J.S.**, Design and analysis of dynamic Huffman codes, *Journal ACM* **34** (1987) 825–845.

[43] **Witten I.H, Neal R.M., Cleary J.G.,** Arithmetic coding for data compression, *Comm. ACM* **30** (1987) 520–540.

[44] **Ziv J., Lempel A.,** A universal algorithm for sequential data compression, *IEEE Trans. on Inf. Th.* **IT–23** (1977) 337–343.

[45] **Ziv J., Lempel A.,** Compression of individual sequences via variable-rate coding, *IEEE Trans. on Inf. Th.* **IT–24** (1978) 530–536.

Prof. Abraham Bookstein
Center for Information
and Language Studies
University of Chicago
1100 E. 57-th Str.
Chicago, IL 60637
USA

Dr. Shmuel T. Klein
Department of Mathematics
and Computer Science
Bar-Ilan University
Ramat-Gan 52900
Israel