

PIECEWISE ARITHMETIC CODING

Jukka Teuhola and Timo Raita

Dept. of Computer Science, Univ. of Turku
Lemminkäisenkatu 14 A
SF-20520 Turku, FINLAND
E-mail: raita@cs.utu.fi

EXTENDED ABSTRACT

1. Introduction

In the *source encoding* problem we are given a set of symbols (and their probabilities), and a coding scheme is to be devised which minimizes the length of the source message. It is well known that *arithmetic coding* [Gua80, RL79] is optimal, i.e. it can get arbitrarily close to the entropy of the source. From a practical point of view, however, there are some problems with the method. First, it produces one codeword for the entire message being transmitted. Therefore the code is *not partially decodable* - we always have to start from the beginning in order to recover a portion of the encoded message. As a consequence, the code is *not indexable* - we cannot say precisely, where in the code a specific source character is represented.

Another problem with arithmetic coding is its *vulnerability*. One-bit errors, either changes (amplitude errors) or omissions (phase errors), usually result in totally scrambled data for the rest of the source message (it is very likely that also the length of the original string changes). If the coding scheme uses fixed length codewords, it cannot recover from phase errors. However, an amplitude error is restricted in the codeword the error occurs. With variable length codewords both errors can propagate very far. On the other hand, e.g. experiments with Huffman coding [Huf52] have shown [LH87] that it can recover rather quickly in practice although no upper bound on the length of the disrupted sequence can be given. This is called the *self-synchronizing* property of the coding scheme.

The ending of the whole code string can be expressed in arithmetic encoding in two ways: First, we can express the length of the original string, which is acceptable in an *off-line* case where the whole string is available at the beginning of the coding process. Second, we can use a special end symbol, which applies also to the *on-line* case. Independently of the method chosen, the end of the code string can be detected only by decoding the whole string, which may be unacceptable in some situations.

The last practical viewpoint is the complicacy of the method itself (see e.g. [WNC87]), making it prone to programming errors and also slower than the simpler, non-optimal coding techniques.

We aim to make improvements in all of the above points by modifying arithmetic coding to produce fixed-length codewords. The idea is to apply arithmetic coding repeatedly for substrings of the original message, so that one substring produces a (maximal) code which fits in a block of predetermined length. The new coding technique, *FIXARI*, is a mixture of block and nonblock coding, but for a restricted codeword (a machine word, say) the problems of nonblock codes become insignificant. The method is non-optimal, but using longer codewords we get closer to the entropy.

Of the related work we mention the coding methods in [Tun68] and [CCL72]. The former aims at creating variable length strings which are close to equiprobable, and the number of which is $\leq 2^{\text{codeword length}}$. The latter is rather similar, but tries to take the dependencies between the characters into account. *FIXARI* has the same idea as Tunstall codes, but makes the probabilities of the resulting substrings more even and generates the codewords on-line (no separate codeword formation phase is needed).

2. Coding method

We shall do arithmetic coding in a small scale, but repeatedly. One coding phase continues until there are no more bits available in the current block of length k reserved for each codeword. Due to the fact that all k -bit blocks are codewords, we identify codewords with blocks of length k in what follows. The result of the arithmetic coding is (conceptually) an

arbitrarily small real interval within $[0,1)$ representing the original message. Since we use fixed-length codewords, each substring is represented as an integer from the range $[0, 2^k-1]$. Still, the basic idea is the same: divide the range recursively in the proportion of character probabilities. Thus each character gets its fair share from the range, embodying in the number of bits required by the character.

Above we mentioned the ending problem of arithmetic coding. Now it is far more serious, because we cannot afford an end character for each codeword. The solution is to modify the basic idea as illustrated in Fig. 1, where the source alphabet is $\{a, b, c\}$ with probabilities $\{0.6, 0.3, 0.1\}$, respectively. The smallest integer in each range is reserved for a special purpose: it expresses an ending substring. The end of the codeword is encountered when the range of the next character falls below one (smallest representable range in the integer scale). This technique applies also at the end of the encoding where the last block may be 'partially filled'. Codeword value zero represents a null string, and is used to express the end of the whole code.

It is obvious that the method is non-optimal: Each possible codeword is not used, because we cannot find a single substring which it represents. Another way to express this is to say that the subrange which causes the next 'underflow' cannot be utilized.

3. Implementation

The above ideas can easily be formulated to the following encoding algorithm. The basic choice was to use unsigned integers (i.e. machine words) as codewords, because this simplifies computations. The following notations are used in the algorithm description:

- Prob(c) gives the probability of character c.
- Cum(c) gives the cumulative probability of character c, with respect to the alphabetic order.
- Highvalue denotes the maximum codeword value, i.e. 2^k-1 .

The algorithm can be formulated as follows.

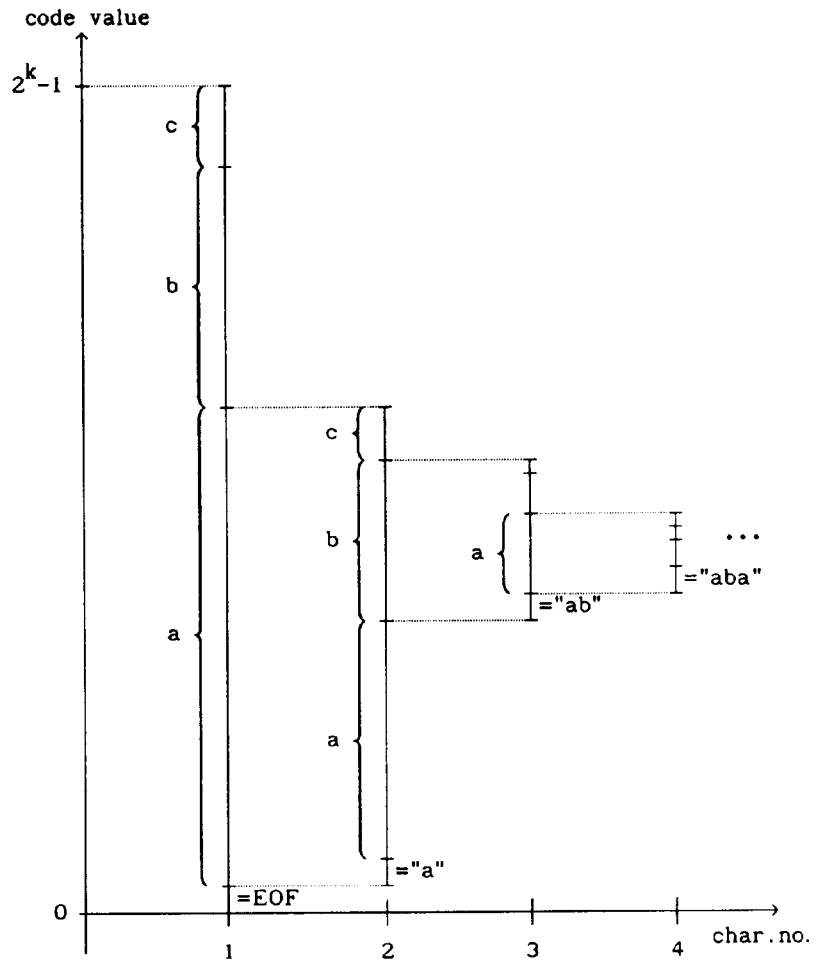


Figure 1. Encoding of substrings "a", "ab", "aba" and end-of-file.

```

begin
  while input string has not been exhausted do
  begin
    c := first character of the unprocessed input;
    low := 0;
    interval := highvalue;
    while [interval * Prob(c)] ≥ 1 do
    begin
      low := low + [interval * Cum(c)] + 1;
      interval := [interval * Prob(c)] - 1;
      read the next source character c,
      at eof set interval := 0
    end;
    output the next codeword, i.e. the value of low;
  end;
  output the zero codeword;
end;

```

We must be very careful with the precision used in the calculations. We use real (floating point) numbers, but in many machines reals are less precise than integers. The problem can be avoided by calculating the lower and upper bounds of the current range as *increments* from the previous lower bound. Now it is precise enough to represent the probabilities of characters as real numbers. Hereby we also get an improvement to the stopping criterion: If

$$[\text{increment of lower bound}] = [\text{increment of upper bound}]$$

then the codeword is full. This means that the real interval can in many cases be < 1 , but truncation of increments makes it = 1. This enables the coding of still one character.

The decoding algorithm is rather obvious. Given a codeword, we can determine the first character from the cumulative probabilities (using e.g. binary search). Then we can reduce the interval as before and repeat the process, until the interval gets < 1 .

4. Analysis

Let us analyse the coding efficiency of FIXARI. This is done by estimating how many codewords from the range $0..2^k-1$ remain unused, measured as bits which are wasted (so-called *loss*).

Actually the loss involves two components. First, besides reserving codewords for each substring with probability close to $1/2^k$, we reserve also codewords for each proper prefix of such a substring. However, a prefix is never coded if it can be extended with any text character (except possibly for once, at the end of the coding process). This loss can be shown to be marginal (fractions of bits per character), and it is neglected in the following. Second, there is a loss from the intervals which are too small to be utilized, i.e. the next subinterval would be < 1 . This loss is more serious, because such an interval exists at the end of each branch, and therefore we analyse it more precisely.

The situation is illustrated in Fig. 2. The length of the code for character c is determined by its information, i.e. by the quantity (real number) $-\log \text{Prob}(c)$. These values are summed up for successive characters, until the next one is too large to fit in the block. The length of the unused part of the codeword should be estimated.

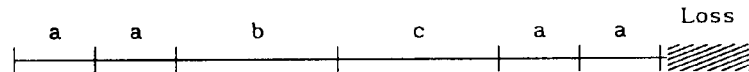


Figure 2. Schematic codeword when encoding "aabcaab..." with $\text{Prob}(\text{"a"}) = 0.5$, $\text{Prob}(\text{"b"}) = \text{Prob}(\text{"c"}) = 0.25$.

The loss is dependent on the distribution of characters. We assume the source to be a natural language text, where the distribution of characters in decreasing order of probability (x) can be estimated by the negative exponential distribution (see [BWC90])

$$F(x) = 1 - e^{-ax}$$

where a is a small constant. (The often suggested Zipfian distribution corresponds to the occurrences of words; it does not seem to hold for individual characters). From F we derive the distribution G of the number of bits needed to represent characters, corresponding to the lengths (y) of the line segments in Fig. 2:

$$G(y) = 1 - \frac{1}{a} 2^{-y}, \text{ when } y \geq -\log_2 a$$

$$= 0, \text{ otherwise}$$

We can assume that the codeword boundary cuts the sequence of line segments at a random point. This assumption is valid, if the codeword is "long enough". The distribution L of the loss (z) is thus the distribution of an arbitrary cut of a line segment. From [Cin75] we find the general formula for L :

$$L(z) = 1 - \frac{1}{\text{Mean}(y)} \cdot \int_z^{\infty} (1 - G(y)) dy$$

Case 1: $z \leq -\log_2 a$

$$L(z) = 1 - \frac{1}{\text{Mean}(y)} \left(\int_z^{-\log_2 a} dy + \int_{-\log_2 a}^{\infty} \frac{1}{a} 2^{-y} dy \right)$$

which gives

$$L(z) = \frac{\ln 2}{1 - \ln a} z$$

Case 2: $z \geq -\log_2 a$

$$L(z) = 1 - \frac{1}{\text{Mean}(y)} \int_z^{\infty} \frac{1}{a} 2^{-y} dy$$

which gives

$$L(z) = 1 - \frac{1}{a(1 - \ln a)} 2^{-z}$$

Now the estimate for the expected loss is

$$E(z) = \int_0^{\infty} z dL(z)$$

$$= \int_0^{-\log_2 a} \frac{\ln 2}{1 - \ln a} z \, dz + \frac{\ln 2}{a(1 - \ln a)} \int_{-\log_2 a}^{\infty} z 2^{-z} \, dz$$

which, after lengthy calculations, gives

$$E(z) = \frac{a - (\ln 2 + 1) \ln a}{(1 - \ln a) \ln 2}$$

Now we can calculate the average number of lost bits per character (LPC) by sharing the loss between the codeword characters:

$$E(\text{LPC}) = \frac{E(z)}{\left[\frac{k}{\left(\frac{1 - \ln a}{\ln 2} \right)} \right]} \approx \frac{a - (\ln 2 + 1) \ln a}{(\ln 2)^2 k} \quad (1)$$

where $(1 - \ln a)/\ln 2$ is the entropy. Using the value $a = 0.11$ (estimated from [BWC90]) we get 0.25 lost bits per character, which is very close to our observations. Note that the loss per character is inversely proportional to the codeword length k , as expected.

5. Experiments

The coding method was tested against source texts in Pascal, English and a pseudo language with probabilities $P('a') = 0.9$, $P('b') = 0.09$, $P('c') = 0.009$, ..., corresponding to the skew distribution $F(x) \approx 1 - e^{-2.3x}$.

The codeword length was 32 bits, i.e. a 4-byte machine word. The method was compared with arithmetic coding (programmed as in [WNC87]) and Huffman coding, and the results have been gathered in Table I. Note that the code/probability tables are excluded from the result size for all the methods. The execution times, given in Table II, were measured on a VAX-6340.

FIXARI gives the lowest compression gains for normal texts, but is the fastest. For skewer distributions FIXARI can be better than Huffman coding. For the selected pseudo text, formula (1)

gives $LPC \approx 0.058$, which is close to the observations. The upper bound for the deviation from entropy is for Huffman $\approx p_1 + 0.086$, where p_1 is the highest probability, see [Gal78]. The decoding time for FIXARI is approximately three times the encoding time, due to the binary search from the cumulative probability table.

It must be emphasized that FIXARI has been developed for practical purposes, in order to get rid of the deficiencies of arithmetic, and also Huffman coding.

Table I

The coding efficiency of the algorithms (bits per character)

Source	Entropy	FIXARI	Arithmetic	Huffman
English	4.61	4.84	4.61	4.64
Pascal	4.24	4.49	4.24	4.29
Pseudo	0.52	0.57	0.52	1.11

Table II

The encoding times of the algorithms (secs/1000 characters)

Source	FIXARI	Arithmetic	Huffman
English	0.04	0.88	0.77
Pascal	0.04	0.95	0.85
Pseudo	0.03	0.22	0.20

6. Conclusions

FIXARI is a practical variant of the arithmetic coding. It is easy to program, fast, and produces fixed-length codewords, enabling partial decoding and indexing of the code. Errors in transmission (bit switches) remain local to the keyword. FIXARI can be made dynamic similarly as other coding methods, but then some of the above mentioned advantages are lost.

References

- [BCW90] Bell, T.C., Cleary, J.G. & Witten, I.H.: Text Compression, Prentice-Hall, 1990
- [Cin75] Cinlar, E.: Introduction to Stochastic Processes, Prentice-Hall, 1975
- [CCL72] Clare, A.C., Cook, E.M. & Lynch, M.F.: The Identification of Variable-Length, Equifrequent Character Strings in a Natural Language Data Base, The Computer Journal, Vol. 15, No. 3, 1972, pp. 259-262
- [Gal78] Gallager, R.G.: Variations on a Theme by Huffman, IEEE Trans. Inf. Th., Vol. IT-24, No. 6, November 1978, pp. 668-674
- [Gua80] Guazzo, M.: A General Minimum-Redundancy Source-Coding Algorithm, IEEE Trans. Inf. Th., Vol. IT-26, No. 1, January 1980, pp. 15-25
- [Huf52] Huffman, D.A.: A Method for the Construction of Minimum-Redundancy Codes, Proceedings of the IRE, Vol. 40, No. 9, September 1952, pp. 1098-1101
- [LH87] Lelewer, D.A. & Hirschberg, D.S.: Data Compression, Computing Surveys, Vol. 13, No. 3, September 1987, pp. 261-296
- [RL79] Rissanen, J.J. & Langdon, G.G.: Arithmetic Encoding, IBM J. Res. Dev., Vol. 23, No. 2, March 1979, pp. 149-162
- [Tun68] Tunstall, B.P.: Synthesis of Noiseless Compression Codes, Ph.D. Thesis, Georgia Institute of Technology
- [WNC87] Witten, I.H., Neal, R. & Cleary, J.G.: Arithmetic Coding for Data Compression, Comm. ACM, Vol. 30, No. 6, June 1987, pp. 520-540