COMMUNICATIONS IN INFORMATION AND SYSTEMS
Vol. 3, No. 2, pp. 139-152, October 2003

004

FAST CODES FOR LARGE ALPHABETS*

BORIS RYABKO†, JAAKKO ASTOLA‡, AND KAREN EGIAZARIAN§

Abstract. We address the problem of constructing a fast lossless code in the case when the source alphabet is large. The main idea of the new scheme may be described as follows. We group letters with small probabilities in subsets (acting as super letters) and use time consuming coding for these subsets only, whereas letters in the subsets have the same code length and therefore can be coded fast. The described scheme can be applied to sources with known and unknown statistics.

Keywords. fast algorithms, source coding, adaptive algorithm, cumulative probabilities, arithmetic coding, data compression, grouped alphabet.

1. Introduction.

The computational efficiency of lossless data compression for large alphabets has attracted attention of researches for ages due to its great importance in practice. The alphabet of $2^8 = 256$ symbols, which is commonly used in compressing computer files, may already be treated as a large one, and with adoption of the UNICODE the alphabet size will grow up to $2^{16} = 65536$. Moreover, there are many data compression methods where the coding is carried out in such a way that, first input data are transformed by some algorithm, and then the resulting sequence is compressed by a lossless code. It turns out that very often the alphabet of the sequence is very large or even infinite. For instance, the run length code, many implementations of Lempel- Ziv codes, Grammar - Based codes [4, 5] and many methods of image compression can be described in this way. That is why the problem of constructing high-speed codes for large alphabets has attracted great attention by researches. Important results have been obtained by Moffat, Turpin [8, 10, 9, 12, 19, 11] and others [3, 6, 7, 14, 15, 2, 18].

For many adaptive lossless codes the speed of coding depends substantially on the alphabet size, because of the need to maintain cumulative probabilities. The time of an obvious (or naive) method of updating the cumulative probabilities is proportional to the alphabet size N. Jones [3] and Ryabko [14] have independently suggested two different algorithms, which perform all the necessary transitions between individual and cumulative probabilities in $O(\log N)$ operations under $(\log N + \tau)$- bit words, where τ is a constant depending on the redundancy required, N is the alphabet size.

*Received on December 30, 2002; accepted for publication on June 26, 2003. Supported by the INTAS under the Grant no. 00-738.
†Professor, Siberian State University of Telecommunication and Computer Science, Kirov Street, 86, 630102 Novosibirsk, Russia. E-mail: ryabko@adm.ict.nsc.ru, URL: http://www.ict.nsc.ru/ryabko
‡Professor, Tampere University of Technology, P.O.B. 553, FIN- 33101 Tampere, Finland. E-mail: jta@cs.tut.fi
§Professor, Tampere University of Technology, P.O.B. 553, FIN- 33101 Tampere, Finland. E-mail: karen@cs.tut.fi

Later many such algorithms have been developed and investigated in numerous papers [8, 15, 2, 10, 9].

In this paper we suggest a method for speeding up codes based on the following main idea. Letters of the alphabet are put in order according to their probabilities (or frequencies of occurrence), and the letters with probabilities close to each others are grouped in subsets (as new super letters), which contain letters with small probabilities. The key point is the following: equal probability is ascribed to all letters in one subset, and, consequently, their codewords have the same length. This gives a possibility to encode and decode them much faster than if they are different, since each subset of the grouped letters is treated as one letter in the new alphabet, whose size is much smaller than the original alphabet. Such a grouping can increase the redundancy of the code. It turns out, however, that a large decrease in the alphabet size may cause a relatively small increase in the redundancy. More exactly, we suggest a method of grouping for which the number of the groups as a function of the redundancy (δ) increases as $c(\log N + 1/\delta) + c_1$, where N is the alphabet size and c, c_1 are constants.

In order to explain the main idea we consider the following example. Let a source generate letters $\{a_0, \dots, a_4\}$ with probabilities $p(a_0) = 1/16$, $p(a_1) = 1/16$, $p(a_2) = 1/8$, $p(a_3) = 1/4$, $p(a_4) = 1/2$, correspondingly. It is easy to see that the following code

$$code(a_0) = 0000, code(a_1) = 0001, code(a_2) = 001, code(a_3) = 01, code(a_4) = 1$$

has the minimal average codeword length. It seems that for decoding one needs to look at one bit for decoding a_4, two bits for decoding a_3, 3 bits for a_2 and 4 bits for a_1 and a_0. However, consider another code

$$\widetilde{code}(a_4) = 1, \widetilde{code}(a_0) = 000, \widetilde{code}(a_1) = 001, \widetilde{code}(a_2) = 010, \widetilde{code}(a_3) = 011,$$

and we see that, on the one hand, its average codeword length is a little larger than in the first code (2 bits instead of 1.825 bits), but, on the other hand, the decoding is simpler. In fact, the decoding can be carried out as follows. If the first bit is 1, the letter is a_4. Otherwise, read the next two bits and treat them as an integer (in a binary system) denoting the code of the letter (i.e. 00 corresponds a_0, 01 corresponds a_1, etc.) This simple observation can be generalized and extended for constructing a new coding scheme with the property that the larger the alphabet size is, the more speeding-up we get.

In principle, the proposed method can be applied to the Huffman code, arithmetic code, and other lossless codes for speeding them up, but for the sake of simplicity, we will consider the arithmetic code in the main part of the paper, whereas the Huffman code and some others will be mentioned only briefly, because, on the one hand, the arithmetic code is widely used in practice and, on the other hand, generalizations are obvious.

The rest of the paper is organized as follows. The second part contains estimations of the redundancy caused by the grouping of letters, and it contains examples for several values of the redundancy. A fast method of the adaptive arithmetic code for the grouped alphabet is given in the third part. Appendix contains all the proofs.

2. The redundancy due to grouping. First we give some definitions. Let $A = \{a_1, a_2, \ldots, a_N\}$ be an alphabet with a probability distribution $\bar{p} = \{p_1, p_2, \ldots, p_N\}$ where $p_1 \geq p_2 \geq \ldots \geq p_N, N \geq 1$. The distribution can be either known a priori or it can be estimated from the occurrence counts. In the latter case the order of the probabilities should be updated after encoding each letter, and it should be taken into account when the speed of coding is estimated. A simple data structure and algorithm for maintaining the order of the probabilities are known and will be mentioned in the third part, whereas here we discuss estimation of the redundancy.

Let the letters from the alphabet A be grouped as follows : $A_1 = \{a_1, a_2, \ldots, a_{n_1}\}$, $A_2 = \{a_{n_1+1}, a_{n_1+2}, \ldots, a_{n_2}\}, \ldots, A_s = \{a_{n_{s-1}+1}, a_{n_{s-1}+2}, \ldots, a_{n_s}\}$ where $n_s = N, s \geq 1$. We define the probability distribution π and the vector $\bar{m} = (m_1, m_2, ..., m_s)$ by

$$\pi_i = \sum_{a_j \in A_i} p_j \tag{1}$$

and $m_i = (n_i - n_{i-1}), n_0 = 0, i = 1, 2, \ldots, s$, correspondingly. In fact,the grouping is defined by the vector $\bar{m}$. We intend to encode all letters from one subset A_i by the codewords of equal length. For this purpose we ascribe equal probabilities to the letters from A_i by

$$\hat{p}_j = \pi_i / m_i \tag{2}$$

if $a_j \in A_i, i = 1, 2, \ldots, s$. Such encoding causes redundancy, defined by

$$r(\bar{p}, \bar{m}) = \sum_{i=1}^{N} p_i \log(p_i/\hat{p}_i). \tag{3}$$

(Here and below $\log(\ ) = \log_2(\ )$.)

The suggested method of grouping is based on information about the order of probabilities (or their estimations). We are interested in an upper bound for the redundancy (3) defined by

$$R(\bar{m}) = \sup_{\bar{p} \in \bar{P}_N} r(\bar{p}, \bar{m});\ \bar{P}_N = \{p_1, p_2, \ldots, p_N\} : p_1 \geq p_2 \geq \ldots \geq p_N\}. \tag{4}$$

The following theorem gives the redundancy estimate.

THEOREM 1. *The following equality for the redundancy (4) is valid.*

$$R(\bar{m}) = \max_{i=1,\ldots,s} \max_{l=1,\ldots,m_i} l \log(m_i/l)/(n_i + l), \tag{5}$$

where, as before, $\bar{m} = (m_1, m_2, ..., m_s), n_i = \sum_{j=1}^{i} m_j, i = 1, ..., s.$

The proof is given in Appendix.

The practically interesting question is how to find a grouping which minimizes the number of groups for a given upper bound of the redundancy δ. Theorem 1 can be used as the basis for such an algorithm. This algorithm is implemented as a Java program and has been used for preparation of all examples given below. The program can be found on the internet and used for practical needs, see

http://www.ict.nsc.ru/˜ryabko/GroupYourAlphabet.html.

Let us consider some examples of such grouping carried out by the program mentioned.

First we consider the Huffman code. It should be noted that in the case of the Huffman code the size of each group should be a power of 2, whereas it can be any integer in case of an arithmetic code. This is because the length of Huffman codewords must be integers whereas this limitation is absent in arithmetic code.

For example, let the alphabet have 256 letters and let the additional redundancy (2) not exceed 0.08 per letter. (The choice of these parameters is appropriate, because an alphabet of $2^8 = 256$ symbols is commonly used in compressing computer files, and the redundancy 0.08 a letter gives 0.01 a bit.) In this case the following grouping gives the minimal number of the groups s.

$$A_1 = \{a_1\}, A_2 = \{a_2\}, \ldots, A_{12} = \{a_{12}\},$$

$$A_{13} = \{a_{13}, a_{14}\}, A_{14} = \{a_{15}, a_{16}\}, \ldots, A_{19} = \{a_{25}, a_{26}\},$$

$$A_{20} = \{a_{27}, a_{28}, a_{29}, a_{30}\}, \ldots, A_{26} = \{a_{51}, a_{52}, a_{53}, a_{54}\},$$

$$A_{27} = \{a_{55}, a_{56}, \ldots, a_{62}\}, \ldots, A_{32} = \{a_{95}, \ldots, a_{102}\},$$

$$A_{33} = \{a_{103}, a_{104}, \ldots, a_{118}\}, \ldots, A_{39} = \{a_{199}, \ldots, a_{214}\},$$

$$A_{40} = \{a_{215}, a_{216}, \ldots, a_{246}\}, A_{41} = \{a_{247}, \ldots, a_{278}\}.$$

We see that each of the first 12 subsets contains one letter, each of the subsets $A_{13}, \ldots, A_{19}$ contains two letters, etc., and the total number of the subsets s is 41. In reality we can let the last subset A_{41} contain the letters $\{a_{247}, \ldots, a_{278}\}$ rather than the letters $\{a_{247}, \dots, a_{256}\}$, since each letter from this subset will be encoded *inside* the subset by 5- bit words (because $\log 32 = 5$).

Let us proceed with this example in order to show how such a grouping can be used to simplify the encoding and decoding of the Huffman code. If someone knows the letter probabilities, he/she can calculate the probability distribution π by (1) and the Huffman code for the new alphabet $\hat{A} = A_1, \dots, A_{41}$ with the distribution π. If we denote a codeword of A_i by $code(A_i)$ and enumerate all letters in each subset A_i from 0 to $|A_i| - 1$, then the code of a letter $a_j \in A_i$ can be presented as the pair of the words

$$code(A_i)\,\{number\,of\,a_j\ \in A_i\},$$

where $\{number\,of\,a_j\ \in A_i\}$ is the $\log |A_i|$- bit notations of the a_j number (inside A_i). For instance, the letter a_{103} is the first in the 16- letter subset A_{33} and a_{246} is the last in the 32- letter subset A_{40}. They will be encoded by $code(A_{33})\,0000$ and $code(A_{40})\,11111$, correspondingly. It is worth noting that the $code(A_i)\,, i = 1, \dots, s$, depends on the probability distribution whereas the second part of the codewords $\{number\,of\,a_j\ \in A_i\}$ does not do that. So, in fact, the Huffman code should be constructed for the 41- letter alphabet instead of the 256- one, whereas the encoding and decoding inside the subsets may be implemented with few operations. Of course, this scheme can be applied to a Shannon code, alphabetical code, arithmetic code and many others. It is also important that the decrease of the alphabet size is larger when the alphabet size is large.

Let us consider one more example of grouping, where the subset sizes don't need to be powers of two. Let, as before, the alphabet have 256 letters and let the additional redundancy (2) not to exceed 0.08 per letter. In this case the optimal grouping is as follows.

$$|A_1| = |A_2| = \dots, |A_{12}| = 1, |A_{13}| = |A_{14}| = \dots = |A_{16}| = 2, |A_{17}| = |A_{18}| = 3,$$

$$|A_{19}| = |A_{20}| = 4, |A_{21}| = 5, |A_{22}| = 6, |A_{23}| = 7, |A_{24}| = 8, |A_{25}| = 9,$$

$$|A_{26}| = 11, |A_{27}| = 12, |A_{28}| = 14, |A_{29}| = 16, |A_{30}| = 19,$$

$$|A_{31}| = 22, |A_{32}| = 25, |A_{33}| = 29, |A_{34}| = 34, |A_{35}| = 39.$$

We see that the total number of the subsets (or the size of the new alphabet) is less than in the previous example (35 instead of 41), because in the first example the subset sizes should be powers of two, whereas there is no such limitation in the second case. So, if someone can accept the additional redundancy 0.01 per bit, he/she can use the new alphabet $\hat{A} = \{A_1, \dots, A_{35}\}$ instead of 256- letter alphabet and implement the arithmetic coding in the same manner as it was described for the Huffman code. (The exact description of the method will be given in the next part). We will not consider the new examples in details, but note again that the decrease in the number of the letters is grater when the alphabet size is larger. Thus, if the alphabet size is 2^{16} and the redundancy upper bound is 0.16 (0.01 per bit), the number of groups s is 39, and if the size is 2^{20} then $s = 40$ whereas the redundancy per bit is the same. (Such calculations can be easily carried out by the above mentioned program).

The required grouping for decreasing the alphabet size is based on the simple theorem 2, for which we need to give some definitions, standard in source coding.

Let γ be a certain method of source coding which can be applied to letters from a certain alphabet A. If p is a probability distribution on A, then the redundancy of γ and its upper bound are defined by

$$\rho(\gamma, p) = \sum_{a \in A} p(a)(|\gamma(a)| + \log p(a)), \quad \hat{\rho}(\gamma) = sup_p \, \rho(\gamma, p), \tag{6}$$

where the supremum is taken over all distributions p, $|\gamma(a)|$ and $p(a)$ are the length of the code word and the probability of $a \in A$, correspondingly. For example, $\hat{\rho}$ equals 1 for the Huffman and the Shannon codes whereas for the arithmetic code $\hat{\rho}$ can be made as small as it is required by choosing some parameters, (see, for ex., [8, 10, 16]). The following theorem gives a formal justification for applying the above described grouping for source coding.

THEOREM 2. *Let the redundancy of a certain code γ be not more than some Δ for all probability distributions. Then, if the alphabet is divided into subsets $A_i, i = 1, \ldots, s$, in such a way that the additional redundancy (3) equals δ, and the code γ is applied to the probability distribution $\hat{p}$ defined by (2), then the total redundancy of this new code γ_{gr} is upper bounded by $\Delta + \delta$.*

The proof is given in Appendix.

Theorem 1 gives a simple algorithm for finding the grouping which gives the minimal number of the groups s when the upper bound for the admissible redundancy (4) is given. On the other hand, a simple asymptotic estimate of the number of such groups and the group sizes can be interesting when the number of the alphabet letters is large. The following theorem can be used for this purpose.

THEOREM 3. *Let $\delta > 0$ be an admissible redundancy (4) of a grouping.*

i) If

$$m_i \le \lfloor \delta \, n_{i-1} \, e / (\log e - \delta \, e) \rfloor, \tag{7}$$

then the redundancy of the grouping $(m_1, m_2, \ldots)$ does not exceed δ, where $n_i = \sum_{j=1}^{i} m_j$, $e \approx 2.718....$).

ii) the minimal number of groups s as a function of the redundancy δ is upper *bounded by*

$$c \log N/\delta + c_1, \quad (8)$$

where c and c_1 are constants and N is the alphabet size, $N \to \infty$.

The proof is given in Appendix.

COMMENT 1. *The first statement of the theorem 3 gives construction of the $\delta-$redundant grouping $(m_1, m_2, ...)$ for an infinite alphabet, because m_i in (7) depends only on previous $m_1, m_2, \ldots, m_{i-1}$.*

COMMENT 2. *Theorem 3 is valid for grouping where the subset sizes $(m_1, m_2, \ldots)$ should be powers of 2.*

3. The arithmetic code for grouped alphabets. Arithmetic coding was introduced by Rissanen [13] in 1976 and now it is one of the most popular methods of source coding. The practically used efficient algorithms of arithmetic code were developed by Moffat [8, 9, 10]. In this part we give first a simplified description of the arithmetic code in order to explain how the suggested method of grouping can be implement along with the arithmetic code.

As before, consider a memoryless source generating letters from the alphabet $A = \{a_1, ..., a_N\}$ with unknown probabilities. Let the source generate a message $x_1 \ldots x_{t-1} x_t \ldots$, $x_i \in A$ for all i, and let $\nu^t(a)$ denote the occurrence count of letter a in the word $x_1 \ldots x_{t-1} x_t$. After first t letters $x_1, \ldots, x_{t-1}, x_t$ have been processed the following letter x_{t+1} needs to be encoded. In the most popular version of the arithmetic code the current estimated probability distribution is taken as

$$p^t(a) = (\nu^t(a) + c)/(t + Nc), a \in A, \quad (9)$$

where c is a constant (as a rule c is 1 or 1/2). Let $x_{t+1} = a_i$, and let the interval $[\alpha, \beta)$ represent the word $x_1 \ldots x_{t-1} x_t$. Then the word $x_1 \ldots x_{t-1} x_t x_{t+1}$, $x_{t+1} = a_i$ will be encoded by the interval

$$[\alpha + (\beta - \alpha)\, q_i^t, \quad \alpha + (\beta - \alpha)\, q_{i+1}^t\,), \quad (10)$$

where

$$q_i^t = \sum_{j=1}^{i-1} p^t(a_j). \quad (11)$$

When the size of the alphabet N is large, the calculation of q_i^t is the most time consuming part in the encoding process. As it was mentioned in the introduction, there are fast algorithms for calculation of q_i^t in

$$T = c_1 \log N + c_2, \quad (12)$$

operations under $(\log N + \tau)$- bit words, where τ is the constant determining the redundancy of the arithmetic code. (As a rule, this length is in proportional to the length of the computer word: 16 bits, 32 bits, etc.)

We describe a new algorithm for the alphabet whose letters are divided into subsets $A_1^t, \ldots, A_s^t$, and the same probability is ascribed to all letters in the subset. Such a separation of the alphabet A can depend on t which is why the notation A_i^t is used. But, on the other hand, the number of the letters in each subset A_i^t will not depend on t which is why it is denoted as $|A_i^t| = m_i$.

In principle, the scheme for the arithmetic coding is the same as in the above considered case of the Huffman code: the codeword of the letter $x_{t+1} = a_i$ consists of two parts, where the first part encodes the set A_k^t that contains a_i, and the second part encodes the ordinal of the element a_i in the set A_k^t. It turns out that it is easy to encode and decode letters in the sets A_k^t, and the time consuming operations should be used to encode the sets A_k^t, only.

We proceed with the formal description of the algorithm. Since the probabilities of the letters in A can depend on t we define in analogy with (1),(2)

$$\pi_i^t = \sum_{a_j \in A_i} p_j, \quad \hat{p}_i^{\,t} = \pi_i^t / m_i \tag{13}$$

and let

$$Q_i^t = \sum_{j=1}^{i-1} \pi_j^t \, . \tag{14}$$

The arithmetic encoding and decoding are implemented for the probability distribution (13), where the probability $\hat{p}_i^t$ is ascribed to all letters from the subset A_i. More precisely, assume that the letters in each A_k^t are enumerated from 1 to m_i, and that the encoder and the decoder know this enumeration. Let, as before, $x_{t+1} = a_i$, and let a_i belong to A_k^t for some k. Then the coding interval for the word $x_1 \ldots x_{t-1} x_t x_{t+1}$ is calculated as follows

$$[\alpha + (\beta - \alpha)(Q_k^t + (\delta(a_i) - 1)\, \hat{p}_i^{\,t}\,), \quad \alpha + (\beta - \alpha)(Q_k^t + \delta(a_i)\, \hat{p}_i^{\,t})\ ), \tag{15}$$

where $\delta(a_i)$ is the ordinal of a_i in the subset A_k^t. It can be easily seen that this definition is equivalent with (10), where the probability of each letter from A_i equals $\hat{p}_i^{\,t}$. Indeed, let us order the letters of A according to their count of occurrence in the word $x_1 \ldots x_{t-1} x_t$, and let the letters in $A_k^t, k = 1, 2, ..., s$, be ordered according to the enumeration mentioned above. We then immediately obtain (15) from (10) and (13). The additional redundancy which is caused by the replacement of the distribution (9) by $\hat{p}_i^{\,t}$ can be estimated using (3) and the theorems 1-3, which is why we may concentrate our attention on the encoding and decoding speed and the storage space needed.

First we compare the time needed for the calculation in (10) and (15). If we ignore the expressions $(\delta(a_i) - 1)\hat{p}_i^t$ and $\delta(a_i)\hat{p}_i^t$ for a while, we see that (15) can be considered as the arithmetic encoding of the new alphabet $\{A_1^t, A_2^t, ..., A_s^t\}$. Therefore, the number of operations for encoding by (15) is the same as the time of arithmetic coding for the s letter alphabet, which by (12) equals $c_1 \log s + c_2$. The expressions $(\delta(a_i)-1)\hat{p}_i^t$ and $\delta(a_i)\hat{p}_i^t$ require two multiplications, and two additions are needed to obtain bounds of the interval in (15). Hence, the number of operations for encoding (T) by (15) is given by

$$T = c_1^* \log s + c_2^*, \tag{16}$$

where c_1^*, c_2^* are constants and all operations are carried out under the word of the length $(\log N + \tau)$- bit as it was required for the usual arithmetic code. In case s is much less than N, the time of encoding in the new method is less than the time of the usual arithmetic code, see (16) and (12).

We focused on the cost of calculating a code and did not take into account the time needed for codeword generation. This time is largely the same for both algorithms.

We describe briefly decoding with the new method. Suppose that the letters $x_1 \dots x_{t-1}x_t$ have been decoded and the letter x_{t+1} is to be decoded. There are two steps required: first, the algorithm finds the set A_k^t with the usual arithmetic code that contains the (unknown) letter a_i. The ordinal of the letter a_i is calculated as follows:

$$\delta() = \lfloor (code(x_{t+1}...) - Q_j^t)/\hat{p}_i^t \rfloor, \tag{17}$$

where $code(x_{t+1}...)$ is the number that encodes the word $x_{t+1}x_{t+2}....$ It can be seen that (17) is the inverse of (15). In order to calculate (17) the decoder should carry out one division and one subtraction. That is why the total number of decoding operations is given by the same formula as for the encoding, see (16).

It is worth noting that multiplications and divisions in (15) and (17) could be carried out faster if the subset sizes are powers of two. But, on the other hand, in this case the number of the subsets is larger, that is why both version could be useful.

So we can see that if the arithmetic code can be applied to an $N-$ letter source, so that the number of operations (under words of a certain length) of coding is

$$T = c_1 \log N + c_2,$$

then there exists an algorithm of coding, which can be applied to the grouped alphabet $A_1^t, \dots, A_s^t$ in such a way that, first, at each moment t the letters are ordered by decreasing frequencies and, second, the number of coding operations is

$$T = c_1 \log s + c_2^*$$

with words of the same length, where c_1, c_2, c_2^* are constants.

As we mentioned above, the alphabet letters should be ordered according to their frequency of occurrences when the encoding and decoding are carried out. Since the frequencies are changing after coding of each message letter, the order should be updated, and the time of such updating should be taken into account when we estimate the speed of the coding. It turns out that there exists an algorithm and data structure (see, for ex.,[8, 10, 18]) , which give a possibility to carry out the updating with few operations per message letter, and the amount of these operations does not depend on the alphabet size and/or a probability distribution. That is why the last estimation of the number of coding operations is valid even if we take into account operations needed for order updating.

It is worth noting that the technique of swapping symbols to keep track of the probability ordering is well known, and has been used, for example, as long ago as dynamic Huffman coding implementation was suggested by Moffat. Since then just about everyone who considers adaptive coding and their changing frequency counts has required a similar mechanism, see for example, [8, 10, 9, 18].

The described method of grouping was applied to arithmetic block code in such a way that a block of several letters was coded using almost the same number of operations as a usual arithmetic code uses for one letter. The preliminary results show that the suggested algorithm essentially speeds up the compression methods, see [17].

4. Appendix. *The proof of Theorem 1.* It is easy to see that the set $\bar{P}_N$ of all distributions which are ordered according to the probability decreasing is convex. Indeed, each $\bar{p} = \{p_1, p_2, \ldots, p_N\} \in \bar{P}_N$ may be presented as a linear combination of vectors from the set

$$Q_N = \{q_1 = (1, 0, \ldots, 0), q_2 = (1/2, 1/2, 0, \ldots, 0), \ldots, q_N = (1/N, \ldots, 1/N) \tag{18}$$

as follows:

$$\sum_{i=1}^{N} i(p_i - p_{i+1}) q_i$$

where $p_{N+1} = 0$.

On the other hand, the redundancy (3) is a convex function, because the direct calculation shows that its second partial derivatives are nonnegative. Indeed, the redundancy (3) can be represented as follows.

$$r(\bar{p}, \bar{m}) = \sum_{i=1}^{N} p_i \log(p_i) \; - \sum_{j=1}^{s} \pi_j (\log \pi_j - \log m_j) =$$

$$\sum_{i=2}^{N} p_i \log(p_i) \; - \sum_{j=2}^{s} \pi_j (\log \pi_j - \log m_j) +$$

$$(1-\sum_{k=2}^{N} p_k)\log(1-\sum_{k=2}^{N} p_k) \;-\; (1-\sum_{l=2}^{s} \pi_l)(\log(1-\sum_{l=2}^{s} \pi_l) - \log m_1).$$

If a_i is a certain letter from A and j is such a subset that $a_i \in A_j$ then, the direct calculation shows that

$$\partial r/\partial p_i = \log_2 e\,(\,\ln p_i - \ln \pi_j - \ln(1-\sum_{k=2}^{N} p_k) + \ln(1-\sum_{l=2}^{s} \pi_l)\,) + constant,$$

$$\partial^2 r/\partial^2 p_i = \log_2 e\,((-1/\pi_j + 1/p_i) + (-1/\pi_1 + 1/p_1)).$$

The last value is nonnegative, because, by definition, $\pi_j = \sum_{k=n_j}^{n_{j+1}-1} p_k$ and p_i is one of the summands as well as p_1 is one of the summands of π_1.

Thus, the redundancy is a convex function defined on a convex set, and its extreme points are Q_N from (18). So

$$sup_{\bar{p}\in \bar{P}_N} r(\bar{p}, \bar{m}) = \max_{q\,\in\, Q_N} r(q, \bar{m}).$$

Each $q \in Q_N$ can be presented as a vector $q = (1/(n_i + l), \ldots, 1/(n_i + l), 0, \ldots, 0)$ where $1 \le l \le m_{i+1}, i = 0, \ldots, s-1$. This representation, the last equality, the definitions (18) , (3) and (4) give (5).

Proof of the theorem 2. Obviously,

$$\sum_{a\in A} p(a)(|\gamma_{gr}(a)| + \log p(a)) =$$

$$\sum_{a\in A} p(a)(|\gamma_{gr}(a)| + \log \hat{p}(a)) + \sum_{a\in A} p(a)(\log(p(a)/\hat{p}(a)). \tag{19}$$

Having taken into account that $p(a)$ is the same for all $a \in A_i$ and $|\gamma_{gr}(a)|$ is the same for all $a \in A_i$, $i = 1, ..., s$, we define $\ddot{p}(i) = p(a), a \in A_i, l(i) = |\gamma_{gr}(a)|, a \in A_i$, for all $i = 1, ..., s$. From those definitions, (1),(2) and (19) we obtain

$$\sum_{a\in A} p(a)(|\gamma_{gr}(a)| + \log \hat{p}(a)) = \sum_{i=1}^{s}(l(i) + \log \ddot{p}(i))(\sum_{a\in A_i} p(a)\,) =$$

$$\sum_{i=1}^{s}(l(i) + \log \ddot{p}(i)) \sum_{a\in A_i} \hat{p}(a) = \sum_{a\in A} \hat{p}(a)(|\gamma_{gr}(a)| + \log \hat{p}(a)).$$

This equality and (19) gives

$$\sum_{a\in A} p(a)(|\gamma_{gr}(a)| + \log p(a)) =$$

$$\sum_{a\in A} \hat{p}(a)(|\gamma_{gr}(a)| + \log \hat{p}(a)) + \sum_{a\in A} p(a)(\log(p(a)/\hat{p}(a)).$$

From this equality, the statement of the theorem and the definitions (3) and (6) we obtain

$$\sum_{a\in A} p(a)(|\gamma_{gr}(a)| + \log p(a)) \leq \Delta + \delta.$$

Theorem 2 is proved.

The proof of the theorem 3. The proof is based on the theorem 1. From (5) we obtain the following obvious inequality

$$R(\bar{m}) \leq \max_{i=1,\ldots,s} \max_{l=1,\ldots,m_i} l\ \log(m_i/l)/n_i. \tag{20}$$

Direct calculation shows that

$$\partial(\log(m_i/l)/n_i)/\partial l = \log_2 e\,(\ln(m_i/l) - 1)/n_i,$$

$$\partial^2(\log(m_i/l)/n_i)/\partial l^2 = -\log_2 e/(l\,n_i) < 0$$

and consequently the maximum of the function $\log(m_i/l)/n_i$ is equal to $m_i \log e/(e\,n_i)$, when $l = m_i/e$. So,

$$\max_{l=1,\ldots,m_i} l\ \log(m_i/l)/n_i \leq m_i \log e/(e\,n_i)$$

and from (20) we obtain

$$R(\bar{m}) \leq \max_{i=1,\ldots,s} m_i \log e/(e\,n_i). \tag{21}$$

That is why, if

$$m_i \leq \delta\, e\, n_i/\log e \tag{22}$$

then $R(\bar{m}) \leq \delta$. By definition (see the statement of the theorem) , $n_i = n_{i-1} + m_i$ and we obtain from (22) the first claim of the theorem. Taking into account that $n_{s-1} < N \leq n_s$ and (21), (22) we can see that, if

$$N = \acute{c}_1(1 + \delta e/\log e)^s + \acute{c}_2,$$

then $R(\bar{m}) \leq \delta$, where $\acute{c}_1$ and $\acute{c}_2$ are constants and $N \to \infty$. Taking the logarithm and applying the well known estimation $\ln(1+\varepsilon) \approx \varepsilon$ when $\varepsilon \approx 0$, we obtain (8). The theorem is proved.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ulman, *The desighn and analysis of computer algorithms*, Reading, MA: Addison-Wesley, 1976.

[2] P. Fenwick, *A new data structure for cumulative probability tables,* Software – Practice and Experience, 24:3(1994) pp. 327–336, (Errata published in vol. 24, no. 7, p. 667, July 1994.)

[3] D. W. Jones, *Application of splay trees to data compression,* Communications of the ACM, 31:8(1988), pp. 996-1007.

[4] J. C. Kieffer and E. H. Yang, *Grammar-based codes: a new class of universal lossless source codes,* IEEE Trans. Inform. Theory, 46:3(2000), pp. 737–754.

[5] J. C. Kieffer, E. H. Yang, G. J. Nelson, and P. Cosman, *Universal lossless compression via multilevel pattern matching,* IEEE Trans. Inform. Theory, 46:4(2000), pp. 1227–1245.

[6] S. T. Klein, *Skeleton trees for the efficient decoding of Huffman encoded texts*, Information Retrieval, 3:1(2000), pp. 7–23.

[7] M. Liddell and A. Moffat, *Hybrid Prefix Code for Practical Use,* In: Procedeengs of IEEE Data Compression Conference, (DCC'2003), 2003, pp. 392–401.

[8] A. Moffat, *Linear time adaptive arithmetic coding,* IEEE Transactions on Information Theory, 36:2(1990), pp. 401–406.

[9] A. Moffat, *An improved data structure for cumulative probability tables,* Software – Practice and Experience, 29:7(1999), pp. 647-659.

[10] A. Moffat, R. Neal, and I. Witten, *Arithmetic Coding Revisited,* ACM Transactions on Information Systems,16:3(1998), pp. 256–294.

[11] A. Moffat and A. Turpin, *On the implementation of minimum redundancy prefix codes,* IEEE Transactions on Communications, 45:10(1997), pp. 1200–1207.

[12] A. Moffat and A. Turpin, *Efficient Construction of Minimum-Redundancy Codes for Large Alphabets,* IEEE Trans. Inform. Theory,44:4(1998), pp. 1650–1657.

[13] J. Rissanen, *Generalized Kraft inequality and arithmetic coding,* IBM J. Res. Dev., 20:5(1976), pp. 198–203.

[14] B. Ya. Ryabko, *A fast sequential code.* Dokl. Akad. Nauk SSSR 306:3(1989), pp. 548–552 (Russian); translation in: Soviet Math. Dokl., 39:3(1989), pp. 533-537.

[15] B. Ryabko, *A fast on-line adaptive code,* IEEE Trans. Inform. Theory, 38:4(1992), pp. 1400–1404.

[16] B. Ryabko and A. Fionov, *Fast and Space-Efficient Adaptive Arithmetic Coding,* in: Proceedings of Cryptography and Coding, 7th IMA International Conference, Cirencester, UK, December 1999. LNCS 1746, pp. 270–279.

[17] B. Ryabko, G. Marchokov, K. Egiazarian, and J. Astola, *The fast algorithm for the block arithmetic code and its applications to image compression,* in: International Workshop "Trends and recent achivments in information technology", 16 - 18 May 2002, Cluj Napoca, Romania, pp. 58–61.

[18] B. Ryabko and J. Rissanen, *Fast Adaptive Arithmetic Code for Large Alphabet Sources with Asymmetrical Distributions,* IEEE Communications Letters, 7:1(2003), pp. 33–35.

[19] A. Turpin and A. Moffat, *On-line adaptive canonical prefix coding with bounded compression loss,* IEEE Trans. Inform. Theory, 47:1(2001), pp. 88- 98.