PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 36/08

Merge Source Coding

Bruno Tenório Ávila

Eduardo Sany Laber

Marcelo Gattass

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900

RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 36/08 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena August, 2008

Merge Source Coding

Bruno Tenório Ávila, Eduardo Sany Laber and Marcelo Gattass

bavila@inf.puc-rio.br, laber@inf.puc-rio.br, mgattass@tecgraf.puc-rio.br

Abstract. This paper presents a new entropy coding called *Merge Source Coding*, which is based on merging algorithms. We show that any merging algorithm can be used as a basis for a binary entropy coder. As a consequence, a new binary entropy coder is proposed, which is called *Binary Merge Coder*. Experimental evaluation shows that it presents little redundancy and that it is much faster than arithmetic coders and quantized indexing coders. We believe that it can be a valuable contribution to those interested in real-time compression.

Keywords: Entropy Coding, Source Coding, Merging Algorithms

Resumo. Este artigo apresenta uma nova codificação de entropia chamada *Codificação Intercalada de Fonte*, na qual é baseada nos algoritmos de intercalação. Nós mostramos que qualquer algoritmo de intercalação pode ser usado como base para um codificador binário de entropia. Como conseqüência, um novo codificador binário de entropia é proposto, na qual é chamado de *Codificador de Intercalação Binária*. Os experimentos mostraram que ele apresenta pouca redundância e que é mais rápido que os codificadores aritméticos e os codificadores de índice quantizados. Nós acreditamos que ele pode ser uma contribuição valiosa para aqueles interessados em compressão em tempo real.

Palavras-chave: Codificação de Entropia, Codificação de Fonte, Algoritmos de Intercalação

In charge for publications:
Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3114-1516 Fax: +55 21 3114-1530
E-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

1 Introduction

An entropy coder is a lossless data compression algorithm that does not use semantic aspects of the data in order to remove its redundancy. Its optimality is achieved according to Shannon's source coding theorem [23]. There are several types of entropy coders that can be categorized according to: the way the symbols' frequencies are updated (e.g. static or adaptive); the block's length size of symbols (eg. fixed or variable); and the ability to model different kinds of sources (e.g. binary, multialphabet, memoryless, Markov, stationary and nonstationary).

Entropy coders play an important role in communication and compression systems. In modern applications (e.g. high-definition video, ultrasonography and satellite images, databases, file systems, scientific visualization), an exponential growth in the data needed to be transmitted and stored has been observed. Therefore, data compression is crucial to reduce their size, as well as performing it in real-time. For those applications, it is reasonable to increase the redundancy of the compressed data as a tradeoff for a smaller compression time.

The two most popular entropy coders are the Huffman coder and the arithmetic coder. The Huffman coder was initially proposed in [10], developed in [8][9][15], improved in [26] and analyzed in [18]. Despite its simplicity and speed, it has several disadvantages, such as the fact that it cannot encode binary sources and that its adaptive version is not very fast. The arithmetic coder was initially proposed in [20], generalized in [21] and popularized in [19]. It has overcome most of the Huffman coder's limitations; for instance, it can encode binary sources and it has smaller redundancy. However, it is fairly slow due to the many operations performed per input symbol, including multiplication and division operations.

Two other entropy coders that deserve some attention are the interval and the enumerative coder. The interval coder [7] keeps track of the last absolute position of each symbol and outputs an integer per input symbol with the relative position of the current symbol and its last occurrence. It is an adaptive coder designed for sources with more than two symbols. This method presents two major drawbacks: it is not optimal and it cannot encode binary sources. The enumerative coder was initially proposed in [16] [5] and improved in [4]. It is an offline method for encoding strings by storing their lexicographic index. The index size is proportional to the entropy size, so it not a practical method. In order to overcame this problem, the quantized indexing coder has been proposed in [25], which constructs a truncated index that fits in a smaller data structure. It also has some disadvantages, such as the fact that it is still static and performs more operations on less frequent input symbols.

This paper presents a new entropy coding called *Merge Source Coding*, which is based on merging algorithms. Our first contribution is to show that any given merging algorithm can be used as an entropy coder. In fact, we show how to combine any set of merging algorithms to obtain an entropy coder. Therefore, the vast literature on merging algorithms (e.g. online and offline algorithms, distributed and parallel strategies, hardware implementations, algorithm analysis, etc.) implies an important consequence: several kinds of entropy coders can be developed for different applications and needs.

Our second contribution consists of selecting a merging algorithm and transforming it into an entropy coder, called *Binary Merge Coder*, which presents several advantages: a) it is asymptotically optimal; b) it has acceptable redundancy; c) it can model data adaptively with a constant number of operations per input symbol; d) it can be implemented without floating point operations, which makes it suitable for hardware implementation; e) and a table with a pre-computed sequence of symbols can be used to speed up the coding process. Our experimental evaluation shows that it is at least three times faster than the quantized indexing coder and twenty times faster than an arithmetic coder with one multiplication per input symbol. As a consequence, the proposed coder can be a valuable contribution to those interested in real-time compression.

This paper is organized as follows: Section 2 presents the Merge Source Coding; Section 3 presents the Binary Merge Coder along with its implementation details and experimental evaluation; finally, in Section 4 we present our conclusions.

2 Merge Source Coding

In this section, we present the *Merge Source Coding*. We noted that there is a relation between merging algorithms and entropy coders and, as a consequence, they can be used to develop entropy coders. The inverse was not explored.

Merging is one of the basic problems in theoretical computer science [14]. Given two disjoint linearly ordered subsets $A = a_1 < ... < a_m$ and $B = b_1 < ... < b_n$, with $m \leq n$, of a linearly ordered set S, the problem consists of determining the linear ordering of their union by means of a sequence of pairwise comparisons between items in A and in B. The measure of complexity of a merging algorithm is the number of comparisons $C(m, n)$, in the worst case, made by the algorithm for input subsets of sizes m and n. An optimal merging algorithm requires $I(m, n) = \lceil log_2 \binom{m+n}{m} \rceil$ comparisons, in the worst case. Note that when $m = 1$, merging degenerates into binary search whose complexity is $\lceil log_2(n + 1) \rceil$.

The worst-case complexity of merging algorithms has been well studied. In 1971, Hwang [12] solved the case for $m = 2$ proving that $C(2, n) = \lceil log_2 7(n + 1)/12 \rceil + \lceil log_2 14(n + 1)/17 \rceil$. In 1972, Hwang and Lin [13] proposed an efficient algorithm, namely *binary merge*, which provides satisfactory results for all values of m and n. In 1973, Hwang and Deutsch [11] developed a merging algorithm that is better than binary merge for small values of m, but not significantly faster. In 1979, Manacher [17] proposed the first significant improvement over binary merge improving it for $n \geq 8m$ by $31m/336$ comparisons. In 1978 (due to publication delay), it was further improved by Christen [1]. It is better than binary merge if $n > 3m$ and uses at least $m/4$ fewer comparisons if $n \geq 4m$, and asymptotically it uses at least $m/3 - o(m)$ fewer comparisons when n/m tends to infinity. On the other hand, this algorithm is worse than binary merge when $n < 3m$. In 1980, Stockmeyer and Yao [24] and Christen [2] obtained the following result: for $m \leq n \leq \lceil 3m/2 \rceil$, then $C(m, n) = m + n - 1$ and the optimal algorithm is the well-known tape-merge (two-way merge) [14]. In 1993, La Vega [6] proposed a simple probabilistic algorithm for merging. It is more efficient than binary merge for $n/m > (\sqrt{5} + 1)/2 \approx 1.618$.

2.1 The General Coding Process

Now, we show how to relate merging algorithms with entropy coders. In fact, we show first how to relate them to binary entropy coders. Then, we show how to combine a set of merging algorithms to work as one entropy coder and how to extend binary entropy coders to model other kinds of sources.

Let x be a source with alphabet $A = \{a_0, a_1, \ldots, a_{d-1}\}$, where d is the cardinality of A.

Let c_k be the count of the symbol a_k in the source x that defines a probability distribution X and $c = \sum_{k=0}^{d-1} c_k$ the size of x. Let $H(X) = \sum_{k=0}^{d-1} \frac{c_k}{c} \log_2 \frac{c}{c_k}$ be the entropy of the source x. Let $x(i)$ be the i^{th} symbol of x from the left to right. Finally, let y be the binary output generated by the entropy coder.

Given a merging algorithm $\mathcal{A}$, we can obtain an entropy coder $\mathcal{B}$ as follows: if x is the binary source to be coded by $\mathcal{B}$, we first construct two linearly ordered sets s_0 and s_1, where $s_0(s_1)$ stores the absolute positions in x containing 0(1). As an example, if $x = 01011100$, then $s_0 = 1, 3, 7, 8$ and $s_1 = 2, 4, 5, 6$. The output y of $\mathcal{B}$ is the result of the comparisons performed by $\mathcal{A}$ when merging s_0 and s_1, that is, the i^{th} bit of y is 0 if, in the i^{th} comparison performed by $\mathcal{A}$, the element of s_0 is smaller than that from s_1; otherwise, the i^{th} bit of y is 1.

The entropy decoder receives the output y as the input parameter. It knows that the elements of s_0 are the positions of the symbols 0 in the source x, as the same for s_1. However, it does not know the position values of each element. These positions are defined by the bits of the output y, which represents the result of the comparisons performed by the merging algorithm. Note that it is the merging algorithm that defines the order and the elements of s_0 and s_1 to be compared. This has to be respected by the decoder. Thus, if the decoder reads a 0 bit, then this means that the element in s_0 is to the left of the element in s_1; otherwise, it is to the right.

The ideas above form the *Merge Source Coding* and lead to our main result: *any merging algorithm can be used to work as a binary entropy coder*. Considering the vast literature on merging algorithms (e.g. online and offline algorithms, distributed and parallel strategies, hardware implementations, algorithm analysis, etc.), this implies an important consequence: several kinds of entropy coders can be developed for different applications and needs.

There are several asymptotically optimal merging algorithms that can be used as binary entropy coders. In fact, if they are used, then the coder will also be asymptotically optimal, in terms of redundancy. In other words, asymptotically optimal merging algorithms with input sizes c_0 and c_1, where $c = c_0 + c_1$, generate a number of comparisons equal to $C_{opt}(c_0, c_1) = O\left(I(c_0, c_1)\right) = k_1 I(c_0, c_1) + k_2$, where k_1 and k_2 are constants. Since $I(c_0, c_1) \leq cH(c_0, c_1)$ (see [3]), then $C_{opt}(c_0, c_1) \leq k_1 cH(c_0, c_1) + k_2 = O(cH(c_0, c_1))$. In order to reduce redundancy, a set of merging algorithms can be combined to form a binary entropy coder. In fact, one could execute a merging algorithm that performs better for any given values of c_0 and c_1.

A merge coder is not limited to memoryless binary sources. The coder can be extended to model other sources by decomposing them into several memoryless binary sources. Multi-alphabet sources with d distinct symbols can be converted into a binary $\lceil \log_2 d \rceil$-Markov source. Then, an m-Markov source can be decomposed into 2^m independent memoryless binary sources. Now, we can just apply the Binary Merge Coder on these several independent memoryless binary sources.

Most merging algorithms use the position of the previous processed element to find the position of the current element. Thus, if these merging algorithms are used to be converted into a binary entropy coder, then the bits sent to the output by the encoder can be interpreted, in binary form, as the relative position of the elements. This method resembles the interval coder [7], because it also codes the relative position of symbols in the source. The Binary Merge Coder presented in the next section is based on the binary merge algorithm [13], which uses this approach.

3 Binary Merge Coder

In this section, we present the *Binary Merge Coder*. The ideas described in section 2 are applied to the binary merge algorithm [13] in order to convert it into the proposed binary entropy coder.

3.1 Definitions

A *static entropy coder* is a two-pass algorithm: first, it counts the symbols' frequencies; second, it encodes the symbols without updating their frequencies. The frequency table has to be stored before compressing the data in the second pass. *A semi-static entropy coder* is basically the same as the static one, but it decreases the frequencies after coding each input symbol. An *adaptive entropy coder* starts the symbols' frequencies with a positive value (e.g. 1) and updates the frequencies after coding each input symbol. The frequency table does not have to be stored.

3.2 Coding Process

First, we consider the semi-static version of the Binary Merge Coder. Later, we show how to use the static and the adaptive models. In the semi-static version, we use *lfs* and *mfs* to denote the least frequent symbol and the most frequent symbol, respectively, in the suffix of the binary source x that has not been processed yet.

Encoding the encoding process for a binary source x initially verifies if c_{mfs} is bigger than c_{lfs} because the symbol *lfs* can become more frequent than the symbol *mfs* in suffix of x that has not been processed. If this is the case, then they are exchanged. Next, it sets $t = \lfloor \log_2 \frac{c_{mfs}}{c_{lfs}} \rfloor$ and $p = 2^t$. A *lfs* symbol is sought in the next p bits of x. If found, a 1 bit is sent to the output y; the position r of the *lfs* symbol is sent to the output using t bits (in fact, the value $r-1$ is coded in standard binary form); c_{mfs} is decreased by $r-1$ and; c_{lfs} is decremented by 1. Otherwise, if not found, then a 0 bit is sent to the output y and c_{mfs} is decremented by p. The process goes back to the first step until c_{lfs} is 0.

#	c_{mfs}	c_{lfs}	t	p	input x	output y
1	14	3	2	4	00010000000100100	111
2	11	2	2	4	0000000100100	1110
3	7	2	1	2	000100100	11100
4	5	2	1	2	0100100	11100111
5	4	1	2	4	00100	11100111110
6	2	0	-	-	00	11100111110

Table 1: Example of the semi-static encoding process.

For example, let $x = 00010000000100100$. Thus, the *mfs* symbol is 0 with $c_{mfs} = 14$ and the *lfs* symbol is 1 with $c_{lfs} = 3$. The next step is to calculate $t = \lfloor \log_2 \frac{14}{3} \rfloor = 2$ and $p = 2^2 = 4$. Next, a *lfs* symbol is found in the position 4, and a 1 bit is sent to the output so that $y = 1$; the position 4 is also sent using 2 bits, or $4 - 1 = 3 = (11)_2$ and concatenated with y so that the current y becomes 111; the values $c_{mfs} = 14 - 3 = 11$

and $c_{lfs} = 3 - 1 = 2$ are updated. A new iteration starts with the *mfs* symbol as the 0 bit, since $c_{mfs} \geq c_{lfs}$. The values $t = \lfloor \log_2 \frac{11}{2} \rfloor = 2$ and $p = 2^2 = 4$ are calculated. Next, as a *lfs* symbol is not found in the next $p = 4$ bits of x, then a 0 bit is sent to y so that it becomes 1110 and the value $c_{mfs} = 11 - 4 = 7$ is updated. The encoding process is summarized in Table 1. The theoretical lower bound for this example is 11 bits and the output generated by the Binary Merge Coder has 11 bits as well.

Decoding since the coding process is static, the decoder initially reads the values of c_{mfs} and c_{lfs} stored in y. Now, y is the parameter input of the decoder and x the output, which will be exactly like the original source by the end of the decoding process.

The decoding process of compressed data y exchanges the symbol c_{mfs} with c_{lfs} and its respective values, only if $c_{mfs} < c_{lfs}$. Next, it sets $t = \lfloor \log_2 \frac{c_{mfs}}{c_{lfs}} \rfloor$ and $p = 2^t$. Next, it reads one bit flag from the input y. If the bit is 0, then it sends p *mfs* symbols to the output x; c_{mfs} is decremented by p. Otherwise, it reads the next t bits and assigns them to r; it sends r *mfs* symbols and one *lfs* symbol to the output and; c_{mfs} is decremented by r and; c_{lfs} is decremented by 1. The process goes back to the first step until c_{mfs} or c_{lfs} is 0. When this process ends, if c_{mfs} is bigger than 0, then c_{mfs} *mfs* symbols are sent to the output.

#	c_{mfs}	c_{lfs}	t	p	input y	output x
1	14	3	2	4	11100111110	0001
2	11	2	2	4	00111110	00010000
3	7	2	1	2	0111110	0001000000
4	5	2	1	2	111110	000100000001
5	4	1	2	4	110	000100000001001
6	2	0	-	-		00010000000100100

Table 2: Example of the semi-static decoding process.

For example, let $y = 11100111110$. The decoder reads the initial values of c_{mfs} and c_{lfs}, which are 14 and 3, respectively. Thus, the *mfs* symbol is 0 and the *lfs* symbol is 1. The first step is to calculate $t = \lfloor \log_2 \frac{14}{3} \rfloor = 2$ and $p = 2^2 = 4$. Next, it reads the first bit, which is 1; it reads the next $t = 2$ bits and decodes them to $r = (11)_2 = 3$; $r = 3$ *mfs* symbols and one *lfs* symbol are sent to the output; the values $c_{mfs} = 14 - 3 = 11$ and $c_{lfs} = 3 - 1 = 2$ are updated. A new iteration is started with the *mfs* symbol as the 0 bit, since $c_{mfs} \geq c_{lfs}$, so there is no need to exchange. Next, the values $t = \lfloor \log_2 \frac{11}{2} \rfloor = 2$ and $p = 2^2 = 4$ are calculated. Since the next bit is 0, then p *mfs* symbols are sent to the output and the value $c_{mfs} = 11 - 4 = 7$ is updated. After all iterations, if c_{mfs} is bigger than 0, then c_{mfs} *mfs* symbols are sent to the output; otherwise c_{lfs} *lfs* symbols are sent. The decoding process is summarized in Table 2.

3.3 Discussion and Analysis

The Binary Merge Coder encodes each least frequent symbol by its relative position since the previous occurrence. At each iteration, a range is estimated within which the next *lfs* symbol is supposed to be and a bit flag is sent to the output to indicate whether the next *lfs* symbol has been found. The range is estimated by assuming the existence of one *lfs* symbol for each c_{mfs}/c_{lfs} of *mfs* symbols. If a *lfs* symbol is found, its position is coded using exactly t bits. Note that if $t = 0$ then $c_{lfs} \approx c_{mfs}$ (e.g. the number of *mfs* symbols is closer to the number of *lfs* symbols). Thus, the entropy is approximately 1 bit per symbol and the bit flags sent are exactly the same bits from the source.

The subsets creation as described in subsection 2.1 does not have to be explicitly made and then applied to the merging algorithm. The cardinality of both subsets c_{lfs} and c_{mfs} is known due to the first pass of the semi-static coder. However, the position values of each element of both subsets are only determined when it encodes the symbols of the source, in the second pass. Note that the binary merge algorithm searches for the correct position of the current element starting at the position of the last previous element. Therefore, previously searched positions can be stored because the following ones will not need them. Thus, the Binary Merge Coder can encode the symbols as soon as it reads them.

The results of the comparisons performed by the binary merge algorithm are sent to the output y. The binary merge algorithm compares the current element of the smallest subset with the element in the relative position p of the biggest subset. The result of this comparison is the bit flag referred in the Binary Merge Coder. If the element is smaller, then a binary search is performed by the binary merge algorithm over those p elements of the biggest subset. The results of the comparisons have the same bits as the r value in binary form using t bits. Note that only the *lfs* symbols have their relative positions coded. The following theorem analyzes the proposed coder in more detail in terms of redundancy.

Theorem 3.1. *The semi-static Binary Merge Coder is asymptotically optimal, in terms of redundancy.*

Proof. Let $BM(m, n)$ denote the complexity of the binary merge algorithm; then, according to [13], $BM(m, n) < I(m, n) + m$. Thus, the Binary Merge Coder outputs $BM(c_0, c_1) + \sigma$ bits, where σ are the bits required to store the symbols' frequencies c_0 and c_1. Assuming $c_0 \leq c_1$, then $BM(c_0, c_1) + \sigma < I(c_0, c_1) + c_0 + \sigma$. According to [3], $BM(c_0, c_1) + \sigma \leq cH(c_0, c_1) + c_0 + \sigma$, where $c = c_0 + c_1$. If we divide everything by c, then the number of bits per symbol $H'(c_0, c_1)$ of the coder is $H'(c_0, c_1) < H(c_0, c_1) + (c_0 + \sigma)/c$ and it follows that $\lim_{c\to\infty} H'(c_0, c_1) < H(c_0, c_1)$. $\square$

The semi-static Binary Merge Coder can be converted into a static coder just by not updating the frequencies after each symbol is coded. It calculates the values of p and t once at the beginning and uses these fixed values during the coding process. It can also be converted into an adaptive one, requiring two minor modifications: the values of c_k are initially set to one and, as the encoder proceeds, the occurrences of the symbols are incremented rather than not decremented. The decoder requires the same modifications: the symbols' frequencies c_k are set initially to one and, as the decoder proceeds, the frequencies are incremented.

3.4 Implementation Details and Experimental Evaluation

The static Binary Merge Coder has a simple implementation. However, the semi-static and adaptive versions have a performance bottleneck: the calculation of variable t at the coding process, which involves a division, a logarithmic function and a floating-point truncation. In order to overcome this problem, two observations can be made to improve considerably the coding performance. First, given an initial value of t, the next value of t will differ by only one; thus, one only needs to test if the ratio is $t - 1$ or $t + 1$. Second, the value of t is only changed periodically. Thus, the c_{mfs} and c_{lfs} limit values that change the value of t can be previously calculated and the above test needs to be done only when the limit values are reached. In practice, these observations result in a multiplication and division-free implementation of the Binary Merge Coder with no floating-point operations. Now, the calculation of t represents merely a small overhead on the coder's performance.

Another improvement can be achieved with the use of tables of pre-coded symbols. The encoding table is mainly a pre-coding of a given block of symbols of the binary source using a given number of ratios c_{mfs}/c_{lfs}(values of t). However, there may be an impractical number of ratios to be stored in the table, so only a small number of ratios are used. The preferred ratio values consist in zero to three. Ratios greater than three mean that the binary source is sparse and there are more blocks of symbols with no *lfs* symbols, so it can be optimized without using any tables. The decoding table follows the same idea, with exception that its blocks of symbols represent the binary source encoded.

Yet another improvement can be applied: the redundancy can be decreased with the use of a combination of optimal merging algorithms that perform better for a specific ratio. In order to avoid performance loss, it can be integrated to the table of pre-coded symbols. For different ratios, the blocks of symbols are coded with a better merging coder and stored in the table. Thus, the use of a combination of different coders is transparent.

File	Size	Entropy	Arithmetic		QIC		BMC		BMC w. table	
			ET	DT	ET	DT	ET	DT	ET	Red.
Text	62979072	0.67	25.1	18.6	3.1	15.2	5.3	5.6	**0.9**	6.9
Calgary	3152896	0.95	37.1	28.4	5.9	25.9	5.5	3.1	**1.8**	4.6
Video	97425978	0.99	37.4	30.4	7.9	29.3	3.6	3.2	**0.8**	0.0

Table 3: Test results of static coders.

An arithmetic coder [22], the quantized indexing coder (QIC) [25], the Binary Merge Coder (BMC) and its version with tables were compared with respect to coding time and redundancy. All coders were implemented using the C language, compiled with Microsoft Visual Studio 2005, and using binary static models. In order to minimize operation system influence in time measurements, all data were copied to memory before coding and the results also were put in memory. They were executed on an Intel Core 2 Duo with 2GB RAM memory. The coders were tested on a text file with pictures, video and on the Calgary corpus grouped into a single file. The test results are summarized in Table 3. The ET and DT abreviations stand for encoding and decoding time, respectively, in nanoseconds per input symbol. The *Red.* abreviation stands for redundancy in percentage of the theoretical limit.

The arithmetic coder and the quantized indexing coder have near zero redundancy and they were not listed in Table 3. The redundancy of the Binary Merge Coder is practically the same for the versions with and without tables, so it is listed only once. The BMC's redundancy is small in relation to the size of the source, because it presents higher redundancy when a greater compression can be achieved and it has small redundancy when only a smaller compression can be achieved.

The arithmetic coding time is very high which represents a problem for practical applications. Although, a multiplication-free and division-free implementation can greatly decrease coding time, it hardly will reach the BMC's coding time due to the number of operations it has to execute. The quantized indexing coder has a better encoding time and it is proportional to the entropy, since it only codes the *lfs* symbols. However, its decoding time is high, because it has to decode all symbols. Another practical disadvantage of the QIC is the fact that it is inherently static, so it has to perform two passes over the source. The Binary Merge Coder presents encoding and decoding times smaller than arithmetic and QI coders, except for the QIC's encoding time. If the tables are used, then the BMC's encoding time is the smallest, having reached an encoding rate of 149.01 MB/s. The encoding time of the BMC with tables is at least three times faster than the QIC's encoding time and twenty times faster than an arithmetic coder with multiplications and divisions.

4 Conclusions and Open Problems

This paper has presented a new entropy coding called *Merge Source Coding*, which is based on merging algorithms. We have shown that any merging algorithm can be used to work as a binary entropy coder. As a consequence, the binary merge algorithm [13] was converted into a binary entropy coder, called *Binary Merge Coder*. Our experimental evaluation has shown that it presents acceptable redundancy and that it is at least twenty times faster than the arithmetic coder and three times faster than the quantized indexing coder. We believe that it can be a valuable contribution for those interested in real-time compression.

The Merge Source Coding proposed herein also suggests further developments. One of them is to convert other merging algorithms to become a merge coder. Another possibility is the construction of a hybrid merge coder in order to reduce redundancy.

Acknowledgments

Thanks are due to Geisa M. Faustino for hers most valuable assistance. The authors are supported by CNPq.

References

[1] C. Christen. Improving the bound on optimal merging. In *Proceedings of the 19th IEEE Symposium on Foundation of Computer Science)*, pages 259–266, 1978.

[2] C. Christen. On the optimality of the straight merging algorithm. Technical Report 296, University of Montreal, 1978.

[3] J. Cleary and I. Witten. A comparison of enumerative and adaptive codes. *IEEE Transactions on Information Theory*, IT-30(2):306–315, March 1984.

[4] T. M. Cover. Enumerative Source Encoding. *IEEE Transactions on Information Theory*, IT-19(1):73–77, January 1973.

[5] L. D. Davisson. Comments on 'Sequence time coding for data compression'. *Proceedings of IEEE*, 54:2010, December 1966.

[6] W. F. de La Vega, S. Kannan, and M. Santha. Two probabilistic results on merging. *SIAM Journal of Computing*, 22(2):261–271, April 1993.

[7] P. Elias. Interval and recency rank source coding: two on-line adaptive variable length schemes. *IEEE Transactions on Information Theory*, IT-33:3–10, January 1987.

[8] N. Faller. An adaptive system for data compression. In *7th Asilomar Conference on Circuits, Systems and Computing*, pages 593–597, 1973.

[9] R. G. Gallager. Variations on a theme by Huffman. *IEEE Transactions on Information Theory*, IT-24:668–674, November 1978.

[10] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, pages 1098–1101, September 1952.

[11] F. K. Hwang and D. N. Deutsch. A class of merging algorithms. *Journal of ACM*, 20:148–159, 1973.

[12] F. K. Hwang and S. Lin. Optimal merging of 2 elements with n elements. *Acta Informatica*, 1:145–158, 1971.

[13] F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly ordered lists. *SIAM Journal of Computing*, 1:31–39, 1972.

[14] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, 1973.

[15] D. E. Knuth. Dynamic Huffman Coding. *Journal of Algorithms*, pages 163–180, 1985.

[16] T. M. Lynch. Sequence time coding for data compression. *Proceedings of IEEE*, 54:1490–1491, October 1966.

[17] G. K. Manacher. Significant improvements to the Hwang-Lin merging algorithm. *Journal of ACM*, 26:434–440, 1979.

[18] R. L. Milidiú, E. S. Laber, and A. A. Pessoa. Improved analysis of the FGK algorithm. *Journal of Algorithms*, 28:195–211, 1999.

[19] A. Moffat, R. M. Neal, and I. H. Witten. Arithmetic coding revisited. *ACM Transactions on Information Systems*, 16(3):256–294, July 1998.

[20] J. J. Rissanen. Generalized kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20(3):198–203, May 1976.

[21] J. J. Rissanen and G. G. Langdon Jr. Arithmetic coding. *IBM Journal of Research and Development*, 23(2):146–162, March 1979.

[22] D. A. Scott. fpaq0s
http://www.cs.fit.edu/˜mmahoney/compression/fpaq0s.cpp, 2006.

[23] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 623–656, July 1948.

[24] P. K. Stockmeyer and F. F. Yao. On the optimality of linear merge. *SIAM Journal of Computing*, 9:85–90, 1980.

[25] R. V. Tomic. Quantized indexing: beyond arithmetic coding. In *Proceedings of the Data Compression Conference (DCC'06)*, March 2006.

[26] J. S. Vitter. Dynamic huffman coding. *ACM Transaction on Mathematical Software*, 15:158–167, June 1989.