Resilient Parameterized Tree Codes for Fast Adaptive Coding

Amir Said
Imaging Systems Laboratory
HP Laboratories Palo Alto

HPL-2004-102
June 7, 2004*

entropy coding,
tree codes,
adaptive coding

This report presents an introduction to efficient adaptive compression using parameterized prefix codes. Advanced adaptive coding techniques can be quite complex because of the need to reliably estimate the probability of a large number of data symbols, each in a number of coding contexts, and then create the codes for each context, and finally code the data. We present practical alternatives with much smaller complexity, which use a pre-defined group of codes with special structure. The adaptive coding process is simplified to estimating which is the best code for a given symbol. This approach is commonly used with Golomb-Rice codes. However, we demonstrate how these codes are quite sensitive to errors in the code-selection process, which are practically unavoidable, due to statistical uncertainty, and the non-stationary nature of real sources. We propose a new family of codes that would have inferior performance if applied to stationary sources, but that are much more "resilient" to errors in the code selection parameters. C++ source code that exemplifies the implementation of the new codes is provided. In addition we propose a combination of those codes with arithmetic coding, in order to exploit the best characteristics of each, and obtain nearly optimal compression, but with complexity (both memory and computation time) much lower than required for arithmetic-only coding.

Resilient Parameterized Tree Codes for Fast Adaptive Coding

Amir Said
HP Laboratories, Palo Alto, CA

1 Introduction

A fundamental process for all data compression applications is called entropy coding. Generally, we model the compression process by first defining a data source that puts out, for integer indexes $i = 0, 1, 2, \ldots$, data symbols S_i belonging to the set $\{0, 1, 2, \ldots, M_i - 1\}$ (source alphabet). In the encoding stage the data symbols are converted to a set of bits[1]. The objective of entropy coding is to minimize of the number of bits required to represent the original data uniquely, without any loss of information [2]. Figure 1 shows a system for entropy encoding and decoding.

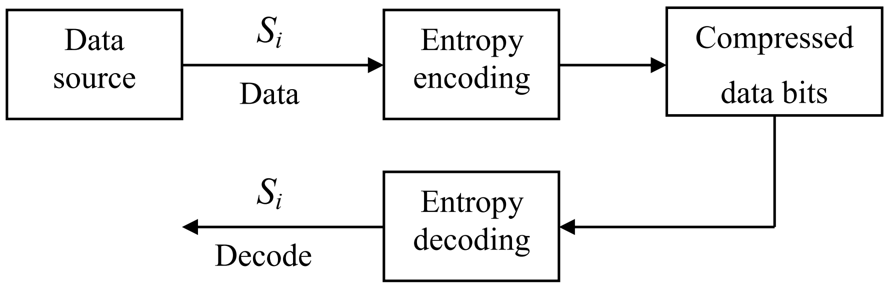

Figure 1 – Entropy coding system.

[1] Other types of output symbols, like decimal digits are possible, but the binary alphabet is by far the most common.

Information theory [2,3] provides the methods to compress the information in an optimal manner. Whatever the coding method, to determine the optimal code for a given data source, we need to have knowledge of the conditional probability of each data symbol, i.e., we need to know the distributions

$$P_i(s) = \text{Prob}\left\{ S_i = s \mid S_{0,}\ S_1, \ldots, S_{i-1} \right\}, \quad i = 0,\ 1, 2, \ldots \tag{1}$$

The optimal number of bits assigned to each data symbol is given by real numbers

$$B_i(s) = \begin{cases} 0, & P_i(s) = 0 \\ \log_2 P_i(s), & P_i(s) > 0 \end{cases} \qquad i = 0,\ 1, 2, \ldots \tag{2}$$

The most commonly used entropy coding methods belong to three categories, explained below.

(a) *Arithmetic coding*. This family of coding methods uses arithmetic operations to assign to each data symbol the optimal fractional number of bits defined by equation (2). Although these methods are quite fast in modern CPUs, the use of arithmetic operations and the management necessary to deal with fractional number of bits makes them in general slower than coding methods that use integer number of bits and table look-up (TLU) [5,7,8].

(b) *Prefix or tree coding*. These coding methods assign an integer number of bits for each coded data symbol. The compression achieved with this strategy is in general sub-optimal, but prefix coding can be computationally quite efficient with TLU for both encoding and decoding. The loss in compression can also be alleviated by combining symbols together [2,3,5].

(c) *String coding*. This group of coding methods includes the Lempel-Ziv methods [2]. They aggregate symbols before coding (forming a “string”), and assigns a fixed or variable integer number of bits for each string. In data sources like English text they can achieve the highest encoding and decoding speeds by working with repeated occurrences of relatively long strings. However, these methods are much less efficient for more random sources, like waveforms (audio, images, video, etc.).

Prefix codes have the property that no ambiguity about the codewords is created by their concatenation. As bits are read sequentially, the decoder always knows when it reaches the end of a codeword and can put out a decoded symbol. These codes are best represented by a tree structure, which guarantees the prefix property, and also permits visual interpretation of the code's properties. Every uniquely-decodable code can be translated into a prefix code with same compression properties [2,3].

Figure 2 shows an example of a binary tree describing a prefix code. In this example, the data alphabet has 8 symbols. In this tree each leaf node corresponds to a data symbol and is represented by a square, while circles represent the other nodes. For each data symbol the encoding process starts in the root node, and moves to the leaf assigned to the current data symbol. The encoder puts out the bits in all the visited branches. For instance, to code symbol "**2**", the encoder puts out the sequence of bits (0, 1, 0). Table I contains the binary codeword for each data symbol in Figure 2. Decoding also starts from the root node. The decoder moves to the right or left branch depending on the bit read, and puts out a decoded symbol when it reaches a leaf. There are several protocols to guarantee that the encoder and the decoder use the same code (tree).

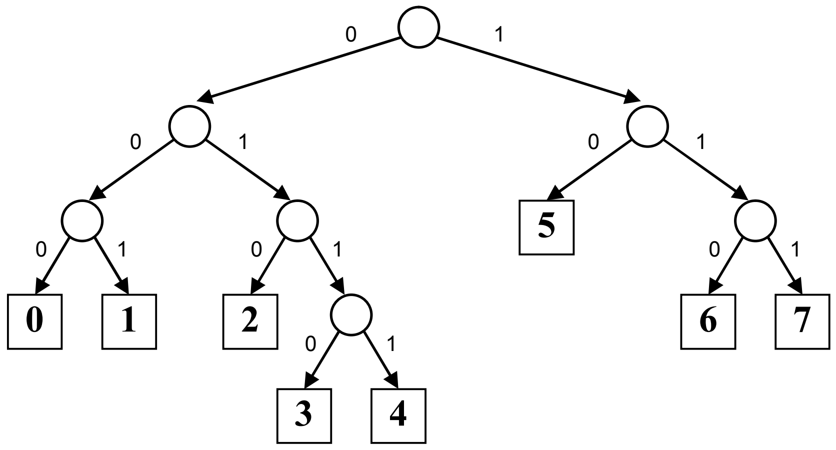

Figure 2 – Tree representing a prefix code.

Table I – Set of codewords defined by the tree of Figure 2 (left-to-right bit order).

Symbol s	**Codeword**	**Bits** $B(s)$
0	000	3
1	001	3
2	010	3
3	0110	4
4	0111	4
5	10	2
6	110	3
7	111	3

2 Adaptive Coding

In practical applications we usually do not know *a priori* the conditional probabilities of the data symbols, as indicated in equation (1), i.e., it is necessary to estimate these probabilities during the encoding and decoding processes. While the notation of equation (1) allows for different probability distributions for each source data index i, in practice we cannot estimate and manage an unbounded number of distributions. In practice we can achieve very good results with a reasonable number of distributions by assuming that symbols in some sets have the same probability distribution. This is equivalent to assume that each source symbol S_n belongs to one *class* of symbols $c \in \{0, 1, 2, \ldots, C-1\}$, depending on alphabet size used for that symbol, and the probability of the symbols in a given class is defined by the set of previously coded symbols[2] $\{S_0, S_1, \ldots, S_{n-1}\}$. Thus, for each class c we assign different probability distributions $P^{(c)}(s)$. A classification function $\Psi_i(S_0, S_1, \ldots, S_{i-1})$ is used to identify the classes, assuming that

$$\Psi_i(S_0, S_1, \ldots, S_{i-1}) = c \quad \Rightarrow \quad \text{Prob}\left\{S_i = s \mid S_0, S_1, \ldots, S_{i-1}\right\} \approx P^{(c)}(s), \tag{3}$$

[2] This set is frequently called the *context* of source symbol S_n.

The classification functions and the number of classes are frequently defined using empirical methods, based on the properties of the data source. The methodology to categorize the source samples that should have the same probability distribution, and to estimate these probabilities, is called *adaptive coding*.

Note that, to follow the commonly accepted definition of adaptive coding, we are not considering *two-pass coding*, in which the encoder first analyzes the complete data sequence $\{S_0, S_1, S_2, \ldots\}$, estimates the probabilities using relative frequencies, computes the optimal codes, and adds the optimal code information to the beginning of the compressed data, and finally, in a second pass, encodes that data. There are several problems with two-pass coding:

(a) It cannot be used for stream coding, i.e., when we want to compress the data as soon as it is available.

(b) The overhead incurred by adding the code information to the compressed data can be quite significant, especially with many classes to better model complex sources (large C) and with large data alphabets (large M).

The notation of equation (3) defines the general case. In practice, the classification functions $\Psi_i(S_0, S_1, \ldots, S_{i-1})$ are computed using only a fixed number r of samples. For example, for one-dimensional sources we commonly have

$$\Psi_i(S_0, S_1, \ldots, S_{i-1}) = \Psi_i(S_{i-r}, S_{i-r+1}, \ldots, S_{i-1}) \qquad (4)$$

i.e., only the last r previously encoded/decoded symbols are used for the classification.

The generalization of equation (4) for 2-dimensional sources (e.g., images) or 3-dimensional sources (e.g., video) requires more complicated notation, so we prefer to use the notation of the most general case in equation (3).

It is possible to improve compression significantly by increasing the number classes C. However, increasing these numbers can degrade compression efficiency for the following reasons:

(a) For best results, the classification of the data samples needs to be based on a large number of data samples (value of r in equation (4)). Keeping track of these samples, and mapping the samples values to a class can be quite demanding.

(b) For each class, it is necessary to continuously estimate the probabilities of all symbols. This task must be repeated many times because estimates improve as more symbols are coded, and because in practical applications the distributions may change in time [7].

(c) For each class and estimated probabilities, it is necessary to compute or update the respective optimal codes.

The problems above, especially (b) and (c), become quite serious when the source alphabet size M is large. For instance, compression can start in a very inefficient manner, because the encoder and decoder need time (many symbols coded) to estimate symbol probabilities reasonably well. When M is large and probabilities are small, the number of symbols coded before good estimates are available can be quite large, and compression efficiency is degraded significantly.

Optimal tree codes are constructed using the algorithm invented by Huffman [2]. The computational complexity to determine an optimal code is proportional to $M \log(M)$ in the worst case, and proportional to M in the best case. This means that when M is large, it is not practical to compute new optimal codes frequently, leading to loss in compression. Some ingenious algorithms were proposed to update the trees automatically, but they were found to be not efficient enough.

Finally, we need to consider that table look-up is very fast only when the tables are not too large (preferably, when they can fit in the CPU's fast cache, or are accessed sequentially). If M and C are large, then the amount of memory to store the estimates, codes, and tables, also becomes prohibitively large.

There are two approaches that can be used to solve these problems:

(a) Assume, for each class, one type of probability distribution (e.g., geometric), and only estimate its parameters. It still may be necessary to compute the optimal codes whenever the parameters change.

(b) Assume some structure for the code trees, defined by a small number of parameters, and only try to estimate the best coding parameters, given the source data context [6].

The advantage of approach (b) is that it completely eliminates the need to estimate probabilities, since their values are implicit in the chosen code. It exploits the fact that source codes normally have little sensitivity to small changes in symbol probabilities. In the next section we present one set of parameterized tree structures (prefix codes).

3 Golomb Codes

Golomb [1] proposed a family of prefix codes that are optimal for a certain common class of data symbol distributions. Each Golomb code is defined uniquely by a positive integer number m. The process of coding a data symbol s is divided in two steps. First, we compute

$$q = \left\lfloor \frac{s}{m} \right\rfloor, \tag{5a}$$

where $\lfloor x \rfloor$ represents the integer part of x. A unary code is used to represent q, i.e., the number q is coded using q 1-bits, followed by a 0-bit.[3]

Next, we compute

$$r = s - mq, \tag{5b}$$

and code $0 \le r < m$ using a number of bits between $\breve{k} = \lfloor \log_2 m \rfloor$ and $\hat{k} = \lceil \log_2 m \rceil$.

[3] An equivalently convention, q 0-bits, followed by a 1-bit, is not used in this work.

We use $G_m(s)$ to represent the binary codeword generated by the Golomb code with parameter *m* for representing data symbol *s*, and we define the operator $\ell[\cdot]$ to represent the number of bits in a codeword (codeword length). Table II shows examples of codewords generated by some Golomb codes. For different values of parameter *m* the columns in Table II contain the codeword $G_m(s)$ and the number of bits $\ell[G_m(s)]$. Figure 3 shows the trees corresponding to the Golomb codes of Table II.

In the special cases when $m = 2^k$, $k = 0, 1, 2, \ldots,$ we can code all possible values of r using exactly k bits, which is quite advantageous in practical applications. These particular codes are also called Rice-Golomb codes, and we represent their codewords by $R_k(s)$, such that,

$$G_{2^k}(s) = R_k(s). \tag{6}$$

In the general case, the number of bits in a codeword is

$$\ell[G_m(s)] = 1 + \left\lfloor \frac{s}{m} \right\rfloor + \begin{cases} \lfloor \log_2 m \rfloor, & \text{if} \quad s - m\lfloor s/m \rfloor < 2^{\lceil \log_2 m \rceil} - m, \\ \lceil \log_2 m \rceil, & \text{otherwise.} \end{cases} \tag{7}$$

In the special case $m = 2^k$, we have

$$\ell[R_k(s)] = 1 + k + q = 1 + k + \left\lfloor \frac{s}{2^k} \right\rfloor. \tag{8}$$

We can see in Equations (7) and (8) that the length of the codewords can only increase with symbol number *s*, meaning that Golomb codes should be used with symbols sorted by decreasing probabilities. In addition, note that Golomb codes are not defined for an alphabet of *M* symbols; instead, they are defined for an infinite number of symbols. This is an advantage when working with large alphabets, and the exact alphabet size is unknown. The codes for the most frequent symbols can be stored in tables, while the codes for the improbable symbols can be generated automatically.

Golomb codes are optimal for sources with geometric distributions in the form

$$P(s) = \mathrm{r}^{s}(1 - \mathrm{r}). \tag{9}$$

The average value of data symbols S for this distribution is

$$E\{S\} = \bar{S} = \frac{\rho}{(1-\rho)}. \tag{10}$$

while the entropy is (in bits)

$$H(\rho) = -\sum_{s=0}^{\infty} P(s)\log_2 P(s) = -\log_2(1-\rho) - \frac{\rho}{1-\rho}\log_2 \rho\ . \tag{11}$$

The optimal value of parameter m for this distribution is

$$m^*(\rho) = \lceil -\log(1+\rho)/\log(\rho) \rceil. \tag{12a}$$

Figure 4 shows the coding performance of Rice-Golomb codes with $k = 0, 1, \ldots, 6$, when used on geometric distribution sources (equations (9) and (10)). Note that the transition between the curves that define the minimum rate occur near the integer values of $\log_2(\bar{S})$, and thus the optimal value of parameter k can be approximated by

$$k^*(\rho) \approx \lfloor \log_2(\bar{S}) \rfloor = \lfloor \log_2(\rho) - \log_2(1-\rho) \rfloor. \tag{12b}$$

Table II – Set of codewords $G_m(s)$ defined by Golomb codes, and their length in bits $\ell[G_m(s)]$,.

Symbol s	$m = 1$		$m = 2$		$m = 3$		$m = 4$	
	Codeword	Bits	Codeword	Bits	Codeword	Bits	Codeword	Bits
0	0	1	00	2	00	2	000	3
1	10	2	01	2	010	3	001	3
2	110	3	100	3	011	3	010	3
3	1110	4	101	3	100	3	011	3
4	11110	5	1100	4	1010	4	1000	4
5	111110	6	1101	4	1011	4	1001	4
6	1111110	7	11100	5	1100	4	1010	4
7	11111110	8	11101	5	11010	5	1011	4
8	111111110	9	111100	6	11011	5	11000	5
9	1111111110	10	111101	6	11100	5	11001	5

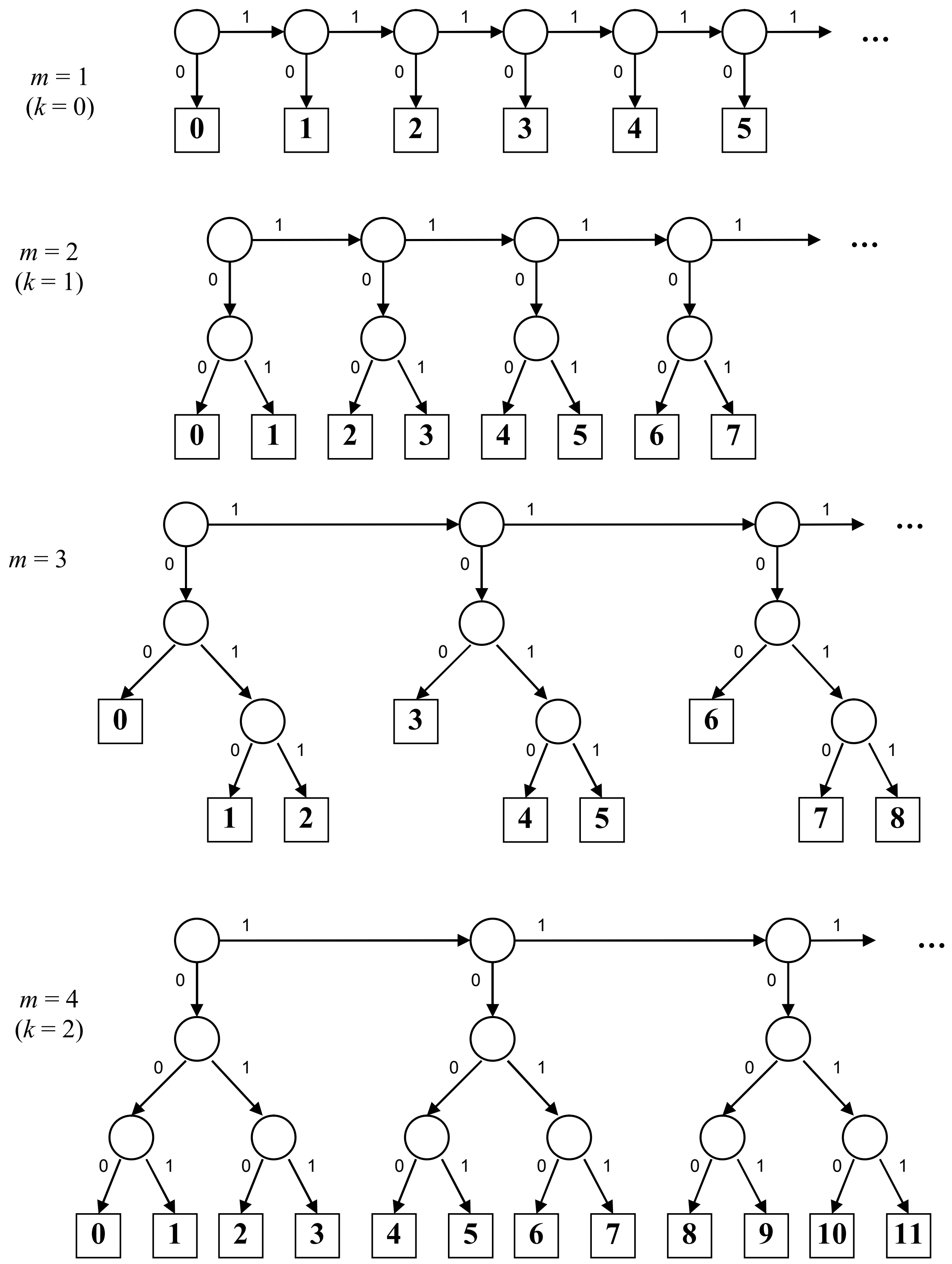

Figure 3 – Trees defining Golomb codes with parameters $m = 1, 2, 3, 4$.

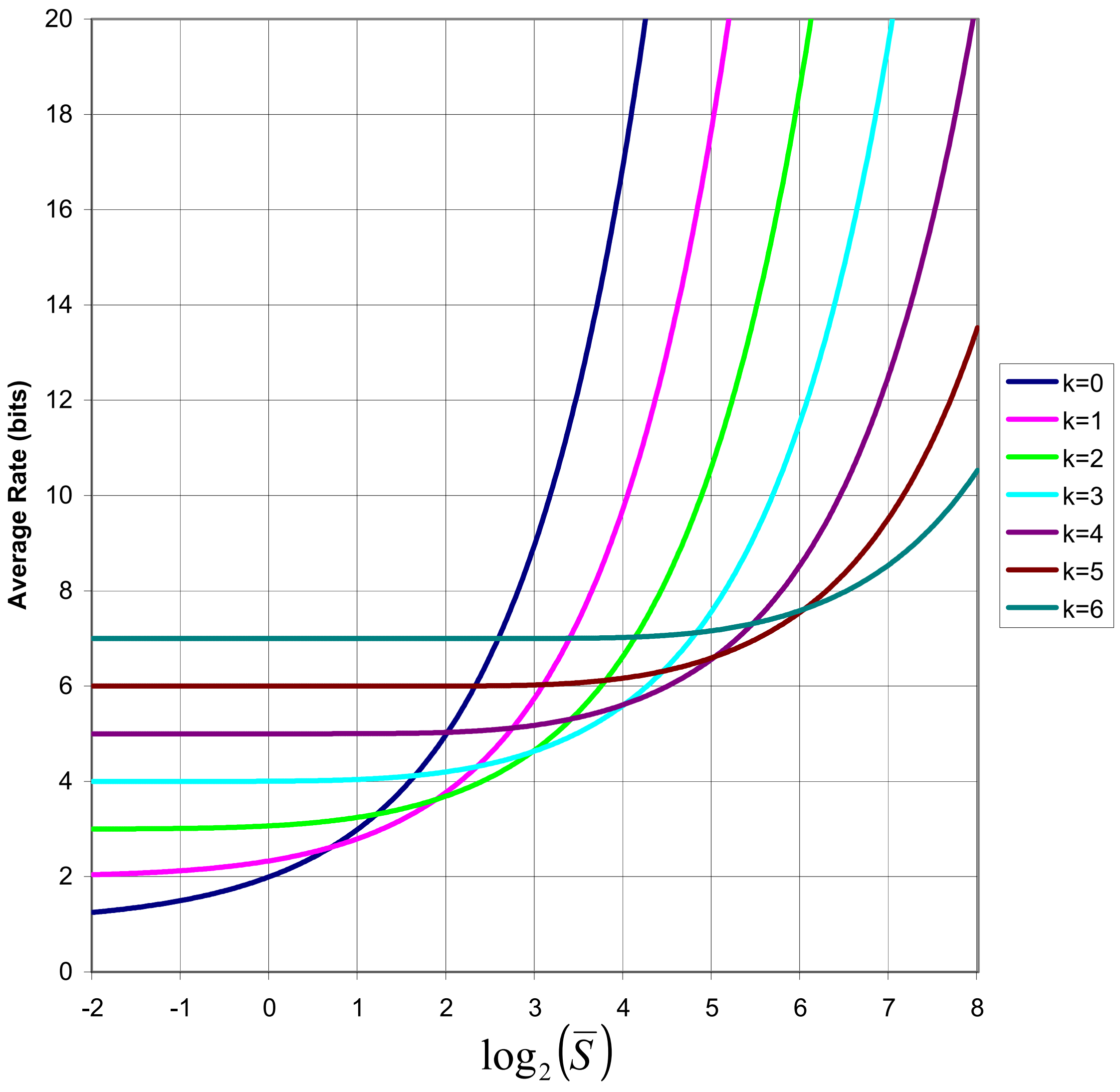

Figure 4 – Coding performance of Golomb-Rice codes for sources with geometric distribution.

In the Appendix A.1 we have a C++ implementation of encoding and decoding functions, which use the parameter *m* to specify the corresponding Golomb code G_m to be used. In the Appendix A.2 we have the same type of functions implementing the corresponding Golomb-Rice codes R_k.

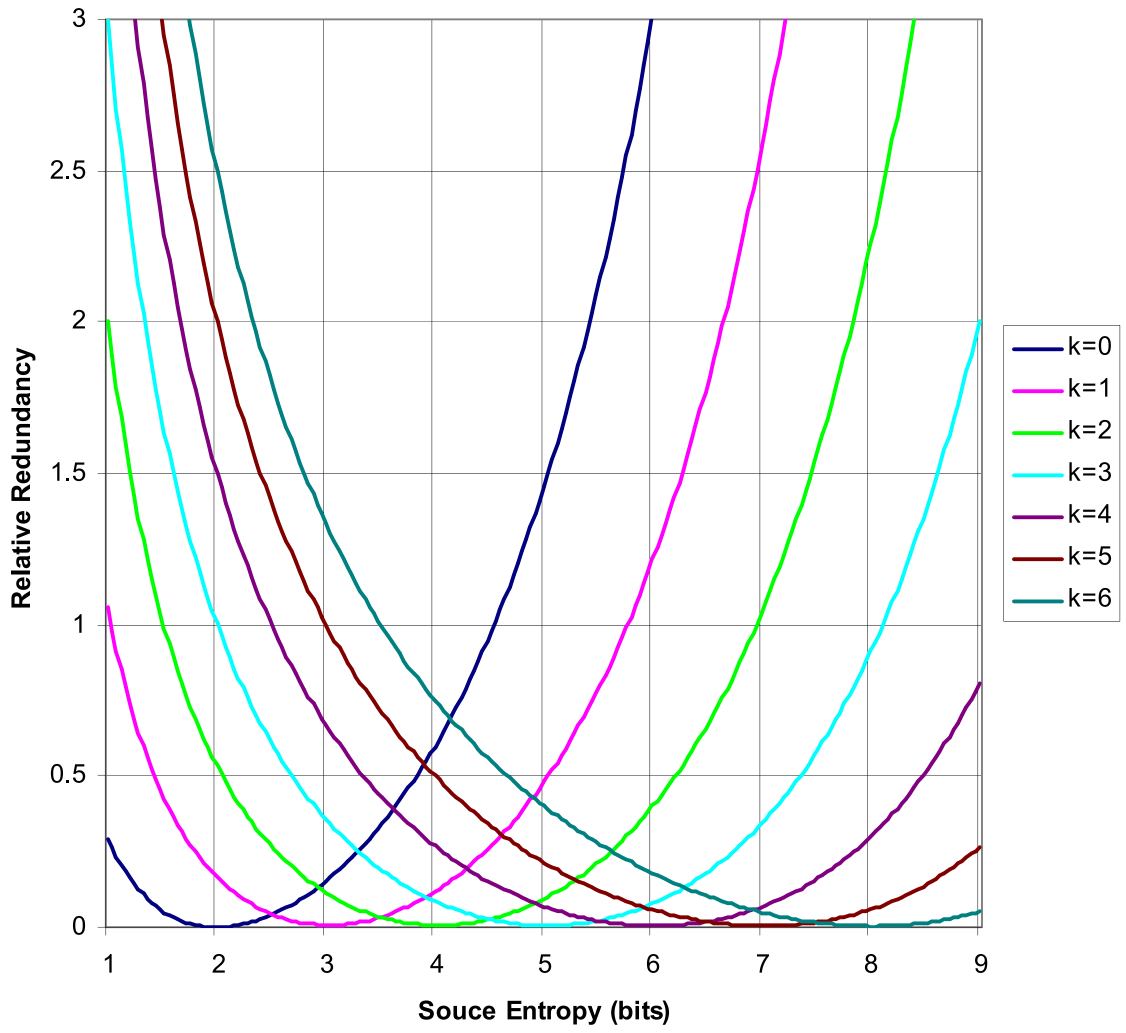

Figure 5 – Relative redundancy of Golomb-Rice codes for sources with geometric distribution.

Defining the relative redundancy of a code as the difference between the average rate and the source entropy, normalized by the source entropy, for the Golomb-Rice codes we have

$$\mathrm{z}_R(k, \mathrm{r}) = \frac{E\{\ell[R_k(s)]\} - H(\mathrm{r})}{H(\mathrm{r})}. \tag{13}$$

Figure 5 shows the relative redundancy of Golomb-Rice codes. The fact that the curves nearly touch the x-axis shows that for some particular values of r shows that these codes can indeed achieve nearly optimal performance in those special cases [9] (the difference is too small to be seen in the graphs).

4 Adaptive Coding with Parameterized Codes

The adaptation strategy to be used with Golomb codes is very simple: we should use a classification function $\Psi_i(S_0, S_1, \ldots, S_{i-1})$ to estimate only what should be the best value of parameter m_i, and then code the symbol with the corresponding Golomb code.

Figure 6 shows a diagram for such form of coding. The encoder is composed of three parts. The first part has a rule for estimating the optimal coding parameter m_i from all previous symbols. The estimated m_i is then used to select the set of codewords to be used to code the current symbol, normally stored in a table. Using the selected table, the encoder copies the bits $G_{m_i}(S_i)$ corresponding to S_i to the compressed bit stream.

We can also consider that in Figure 6 the encoder has codewords organized in a two-dimensional table, and uses as codeword $G_{m_i}(S_i)$ the entry in column m_i and row S_i. Figure 6 also shows the corresponding decoder structure, which must use an optimal parameter estimation that is identical to the one used by the encoder.

In practical applications it is more common to use $m = 2^k$, and instead estimate the best k. This adaptation technique is not indicated for all types of data sources, but for many applications of waveform coding (audio, video, images, etc.), it works quite well, eliminating all the problems associated with estimation of whole probability distributions.

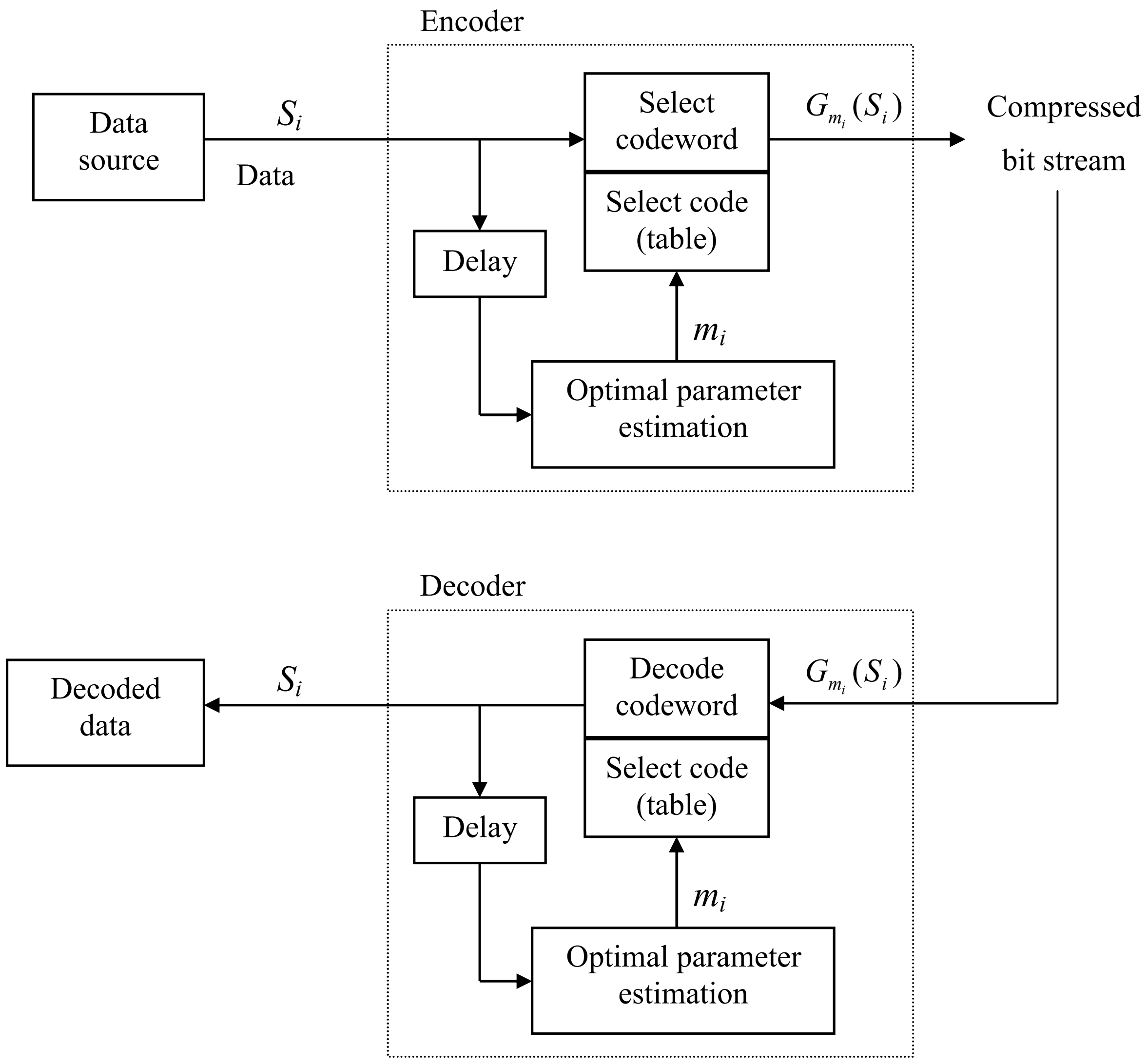

Figure 6 – Adaptive coding with code selected by a single parameter.

5 Resilient Parameterized Tree Codes

A problem with the use of Golomb codes in an adaptive coding strategy is that they are not resilient to errors in the estimation of the code-selection parameter. This fact can be seen in the performance curves in Figure 5: the rates grow too fast around the optimal point, so there is a large penalty (in bits) whenever the parameter *m* is not correctly estimated. For example, suppose the classification function estimates that the best Golomb code parameter is *m* = 1, and $S_i = 300$; then this symbol will be coded using 300 bits equal to 1, followed by a bit equal to 0. This is obviously very inefficient, and also not very smart. From equation (2) we can see that such a codeword was supposed to be used for a symbol with probability $P(S_i) = 2^{-301} \approx 2 \cdot 10^{-91}$. During the encoding process, after encoding a certain number of bit with the unary code, it is possible to infer that it is much more plausible that the parameter *m* had been chosen with a value too small, than assume the occurrence of a data symbol with such astronomically small probability.

The solution we propose to this problem is to change the trees that define the codes.[4] In the Golomb codes the number of symbols (leaf nodes) in the sub-trees that "grow" from the unary[5] code is always equal to its parameter *m*. Our strategy is to keep increasing the number of symbols in those sub-trees, which is in a way similar to changing the parameter *m* of the Golomb code in the middle of a codeword. This way, the number of symbols in the sub-trees is not always *m*, but can be equal to, for example, *m*, *m* +1, *m* +2,…, or equal to *m*, *m*, *m* +1, *m* +1,…, etc. Since we are defining the new codes from their tree structure, it should be clear that they are all tree codes.

What is left to decide is the rule to be used in the creation of these new codes, i.e., how we want to increase the number of symbols in the sub-trees. We call these codes resilient, since they have low sensitivity to errors in the code selection. For our resilient adaptive coding application we consider the two following choices.

(a) **Linear growth codes ($L_{m,d,w}$ codes)**: after *w* consecutive unary-code bits equal to 1, increase the number of symbols in the sub-trees by an amount *d*. We use the symbol *m* to represent the *initial* number of symbols in the trees growing from nodes forming the unary code. The codeword for such code is represented by $L_{m,d,w}(s)$.

[4] A companion report presents a more theoretical justification and analysis.
[5] The unary code is composed of the bits indicated by the horizontal arrows in Figure 3, followed by the first vertical arrow. In fact, we used this special tree layout to stress the importance of those bits.

(b) **Exponential growth codes (**$E_{k,w}$ codes**)**: after w consecutive unary-code bits equal to 1, double the number of symbols in the sub-trees. We assume that the original number of symbols in the trees growing from nodes forming the unary code is $m = 2^k$, so the exponential-growth codes are uniquely defined as $E_{k,w}(s)$.

Figure 7 shows some examples of new tree codes generated using the linear growth strategy. The first example is $L_{m,d,w} = L_{1,1,3}$. Since $m = 1$, the first sub-trees have only one symbol each (0, 1, and 2). After $w = 3$ branches of the unary code the number of symbols is increased by $d = 1$ to two symbols per tree forming pairs (3,4), (5,6), and (7,8). The following three trees (not shown in the figure) would have 3 symbols each, as (9,10,11), (12,13,14), and (15,16,17).

The next example is code $L_{m,d,w} = L_{1,1,1}$, which has the number of symbols per tree growing at each node of the unary code. Thus, the number of symbols grow as 1, 2, 3, …, and the symbols in each tree are (0), (1,2), (3,4,5), (6,7,8,9), (10,11,12,13,15), etc.

Figure 8 shows examples of codes generated using the exponential growth technique. The first example is code $E_{k,w} = E_{0,2}$, which has trees with number of symbols growing exponentially in groups of $w = 2$. Thus, the number of symbols grows as 1, 1, 2, 2, 4, 4, 8, 8, 16, 16, and so on. In the second example the growth is faster because $w = 1$, and the number of symbols grows as 2, 4, 8, 16, 32, 64, …

Note that the Golomb-Rice codes can be considered particular cases of this new types of code, since

$$L_{m,0,w} = L_{m,d,\infty} = G_m, \tag{14}$$

and

$$E_{k,\infty} = R_k. \tag{15}$$

Figure 9 show diagrams with the encoder and decoder for this type of code. Note that the only difference to those in Figure 6 is in the number of parameters to be estimated.

Tables III, IV, V, and VI contain more examples of $L_{m,d,w}$ and $E_{k,w}$ codes. All the examples show that even though the codes are defined by a small set of parameters, there is good deal of diversity left for adaptive coding. Even though the construction of the linear-growth codes is quite simple, the total number of bits for each codeword (bits in the unary code plus bits in the tree following it) cannot be easily described as in Equations (7) or (8). The number of bits in the exponential-growth codes can be computed using the following equations

$$u_{k,w}(s) = \left\lfloor \log_2\left(1 + \frac{s}{2^k w}\right) \right\rfloor \tag{16}$$

$$\ell\left[E_{k,w}(s)\right] = 1 + k - w + (w+1)\, u_{k,w}(s) + \left\lfloor \frac{s + 2^k w}{2^{k+u_{k,w}(s)}} \right\rfloor . \tag{17}$$

The equation for special case $w = 1$,

$$\ell\left[E_{k,1}(s)\right] = 1 + k + 2\left\lfloor \log_2\left(1 + \frac{s}{2^k}\right) \right\rfloor , \tag{18}$$

is particularly interesting because it shows the most significant difference between the new codes and Golomb-Rice codes: comparing Equations (8) and (18), we can see that in the new codes the codeword lengths a proportional to the logarithm of the data values.

Figure 10 shows the relative redundancy of the new exponential-growth codes when applied to geometric distribution sources, using parameter $w = 1$. Note that the curves do not touch the x-axis, meaning that these codes cannot reach optimality for these sources, but on the other hand, when we compare Figure 10 with Figure 5 we can easily see that they are much more resilient to errors in the estimation of parameter k. Figures 11a, 11b, and 12 show the performance of these codes with parameters $w = 2$ and $w = 3$, and how we can trade-off performance for resiliency by changing parameter w.

Appendix sections A.3 and A.4 provide the C++ implementation of the new codes. The first implementation (A.3) uses the parameters (m, d, w) to define the code, and uses the linear growth strategy. The second implementation (A.4) uses the parameters (k, w), and exponential growth.

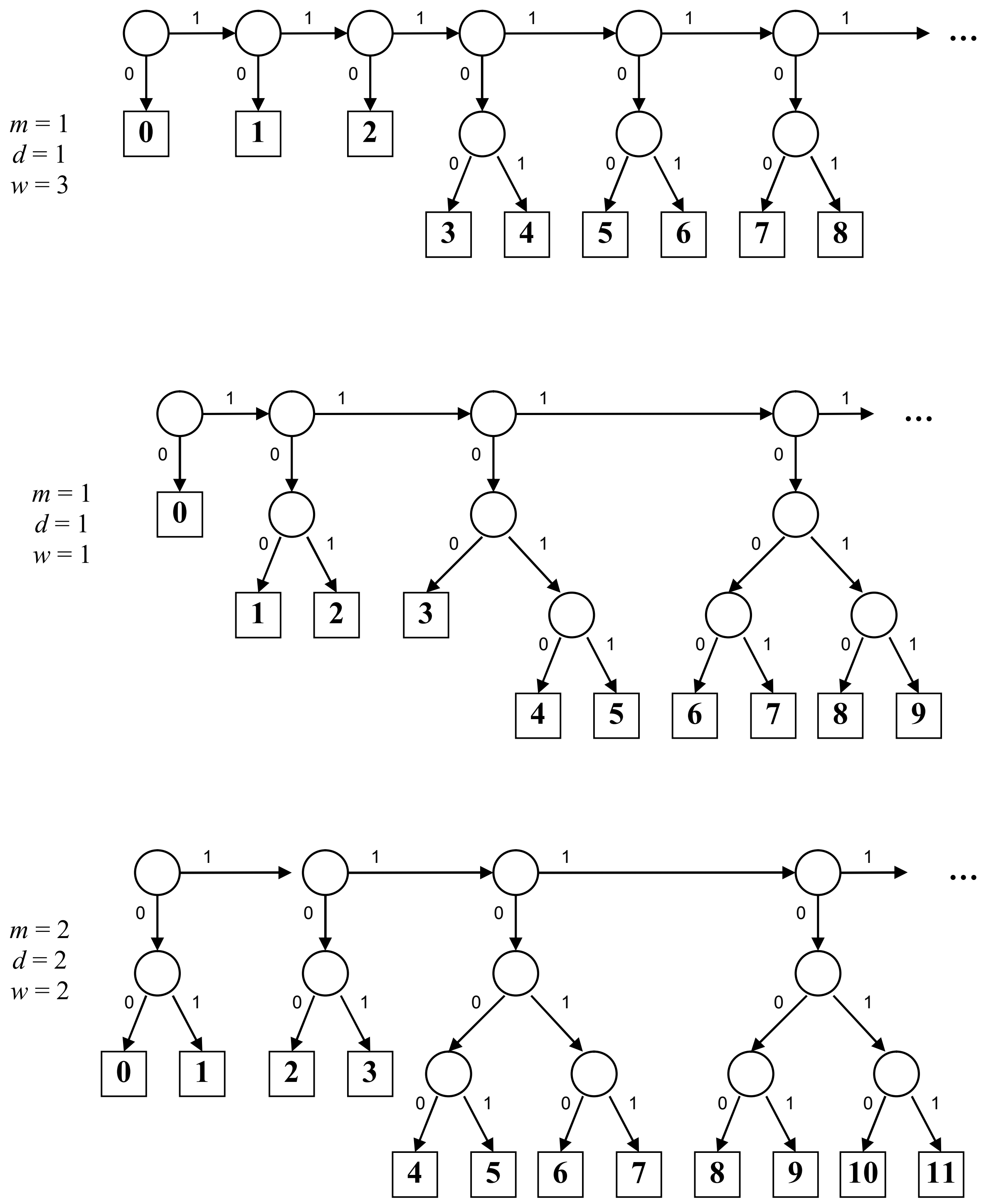

Figure 7 – Examples of resilient tree codes with linear growth.

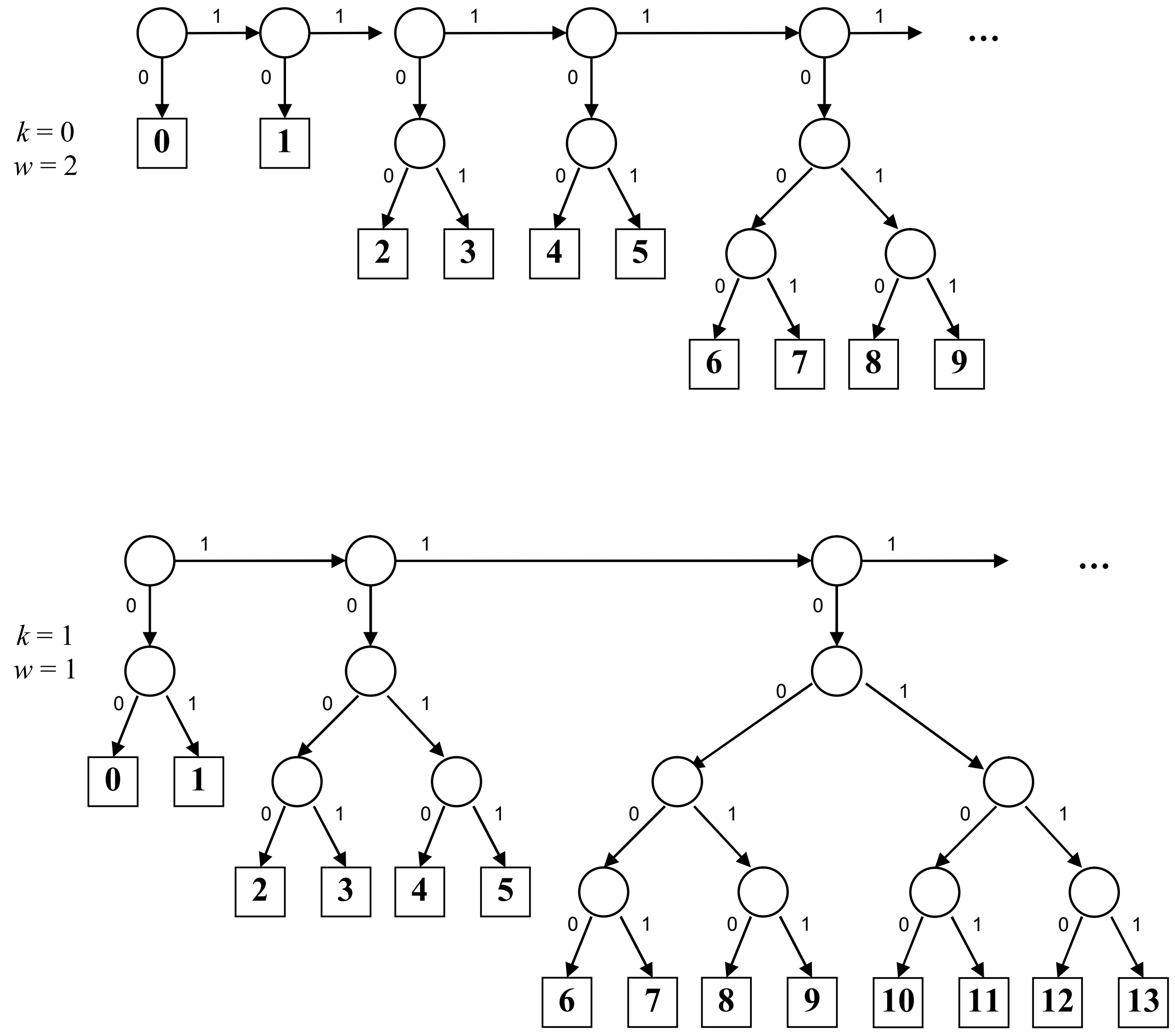

Figure 8 – Examples of resilient tree codes with exponential growth.

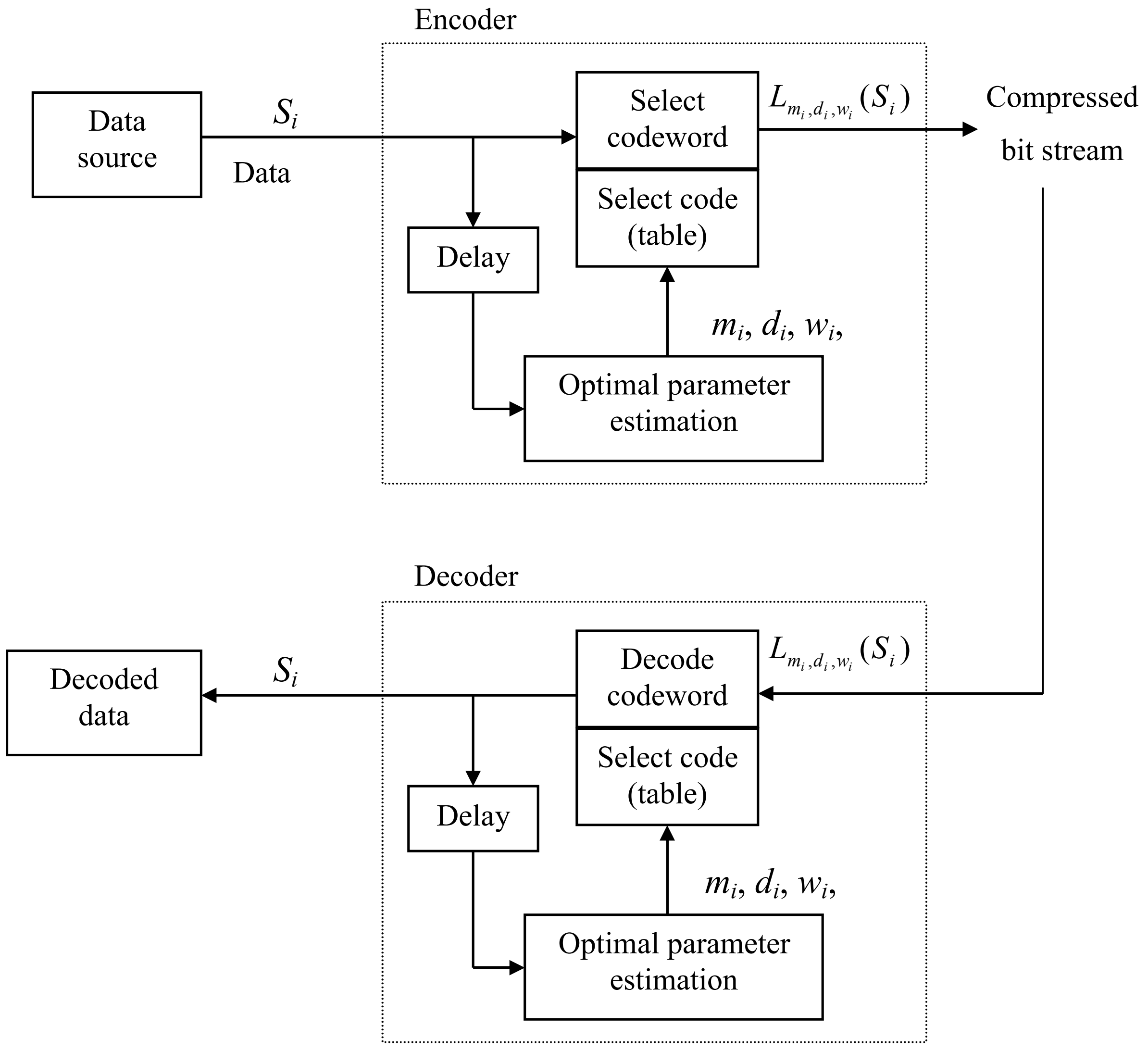

Figure 9 – Adaptive coding with code defined by multiple parameters.

Table III – Set of codewords defined by the linear-growth resilient tree codes with parameters $m = 1$, $d = 1$.

Symbols	m=1, d=1, w=1 ($L_{1,1,1}$)		m=1, d=1, w=2 ($L_{1,1,2}$)		m=1, d=1, w=3 ($L_{1,1,3}$)		m=1, d=1, w=4 ($L_{1,1,4}$)	
	Codewords	Bits	Codewords	Bits	Codewords	Bits	Codewords	Bits
0	0	1	0	1	0	1	0	1
1	100	3	10	2	10	2	10	2
2	101	3	1100	4	110	3	110	3
3	1100	4	1101	4	11100	5	1110	4
4	11010	5	11100	5	11101	5	111100	6
5	11011	5	11101	5	111100	6	111101	6
6	111000	6	111100	6	111101	6	1111100	7
7	111001	6	1111010	7	1111100	7	1111101	7
8	111010	6	1111011	7	1111101	7	11111100	8
9	111011	6	1111100	7	11111100	8	11111101	8
10	1111000	7	11111010	8	111111010	9	111111100	9
11	1111001	7	11111011	8	111111011	9	111111101	9
12	1111010	7	111111000	9	111111100	9	1111111000	10

Table IV – Set of codewords defined by the linear-growth resilient tree codes with parameters $m = 2$, $w = 2$.

Symbols	m=2, d=1, w=2 ($L_{2,1,2}$)		m=2, d=2, w=2 ($L_{2,2,2}$)		m=2, d=3, w=2 ($L_{2,3,2}$)		m=2, d=4, w=2 ($L_{2,4,2}$)	
	Codewords	Bits	Codewords	Bits	Codewords	Bits	Codewords	Bits
0	00	2	00	2	00	2	00	2
1	01	2	01	2	01	2	01	2
2	100	3	100	3	100	3	100	3
3	101	3	101	3	101	3	101	3
4	1100	4	11000	5	11000	5	11000	5
5	11010	5	11001	5	11001	5	11001	5
6	11011	5	11010	5	11010	5	110100	6
7	11100	5	11011	5	110110	6	110101	6
8	111010	6	111000	6	110111	6	110110	6
9	111011	6	111001	6	111000	6	110111	6
10	1111000	7	111010	6	111001	6	111000	6
11	1111001	7	111011	6	111010	6	111001	6
12	1111010	7	1111000	7	1110110	7	1110100	7

Table V – Set of codewords defined by the exponential-growth resilient tree codes ($E_{k,w}$) with parameter $k = 0$.

Symbol s	$k = 0, w = 1$ ($E_{0,1}$)		$k = 0, w = 2$ ($E_{0,2}$)		$k = 0, w = 3$ ($E_{0,3}$)		$k = 0, w = 2$ ($E_{0,4}$)	
	Codewords	Bits	Codewords	Bits	Codewords	Bits	Codewords	Bits
0	0	1	0	1	0	1	0	1
1	100	3	10	2	10	2	10	2
2	101	3	1100	4	110	3	110	3
3	11000	5	1101	4	11100	5	1110	4
4	11001	5	11100	5	11101	5	111100	6
5	11010	5	11101	5	111100	6	111101	6
6	11011	5	1111000	7	111101	6	1111100	7
7	1110000	7	1111001	7	1111100	7	1111101	7
8	1110001	7	1111010	7	1111101	7	11111100	8
9	1110010	7	1111011	7	1111110000	9	11111101	8
10	1110011	7	11111000	8	1111110001	9	111111100	9
11	1110100	7	11111001	8	1111110010	9	111111101	9
12	1110101	7	11111010	8	1111110011	9	11111111000	11

Table VI – Set of codewords defined by the exponential-growth resilient tree codes ($E_{k,w}$) with parameter $k = 1$.

Symbol s	$k = 1, w = 1$ ($E_{1,1}$)		$k = 1, w = 2$ ($E_{1,2}$)		$k = 1, w = 3$ ($E_{1,3}$)		$k = 1, w = 4$ ($E_{1,4}$)	
	Codewords	Bits	Codewords	Bits	Codewords	Bits	Codewords	Bits
0	00	2	00	2	00	2	00	2
1	01	2	01	2	01	2	01	2
2	1000	4	100	3	100	3	100	3
3	1001	4	101	3	101	3	101	3
4	1010	4	11000	5	1100	4	1100	4
5	1011	4	11001	5	1101	4	1101	4
6	110000	6	11010	5	111000	6	11100	5
7	110001	6	11011	5	111001	6	11101	5
8	110010	6	111000	6	111010	6	1111000	7
9	110011	6	111001	6	111011	6	1111001	7
10	110100	6	111010	6	1111000	7	1111010	7
11	110101	6	111011	6	1111001	7	1111011	7
12	110111	6	11110000	8	1111010	7	11111000	8

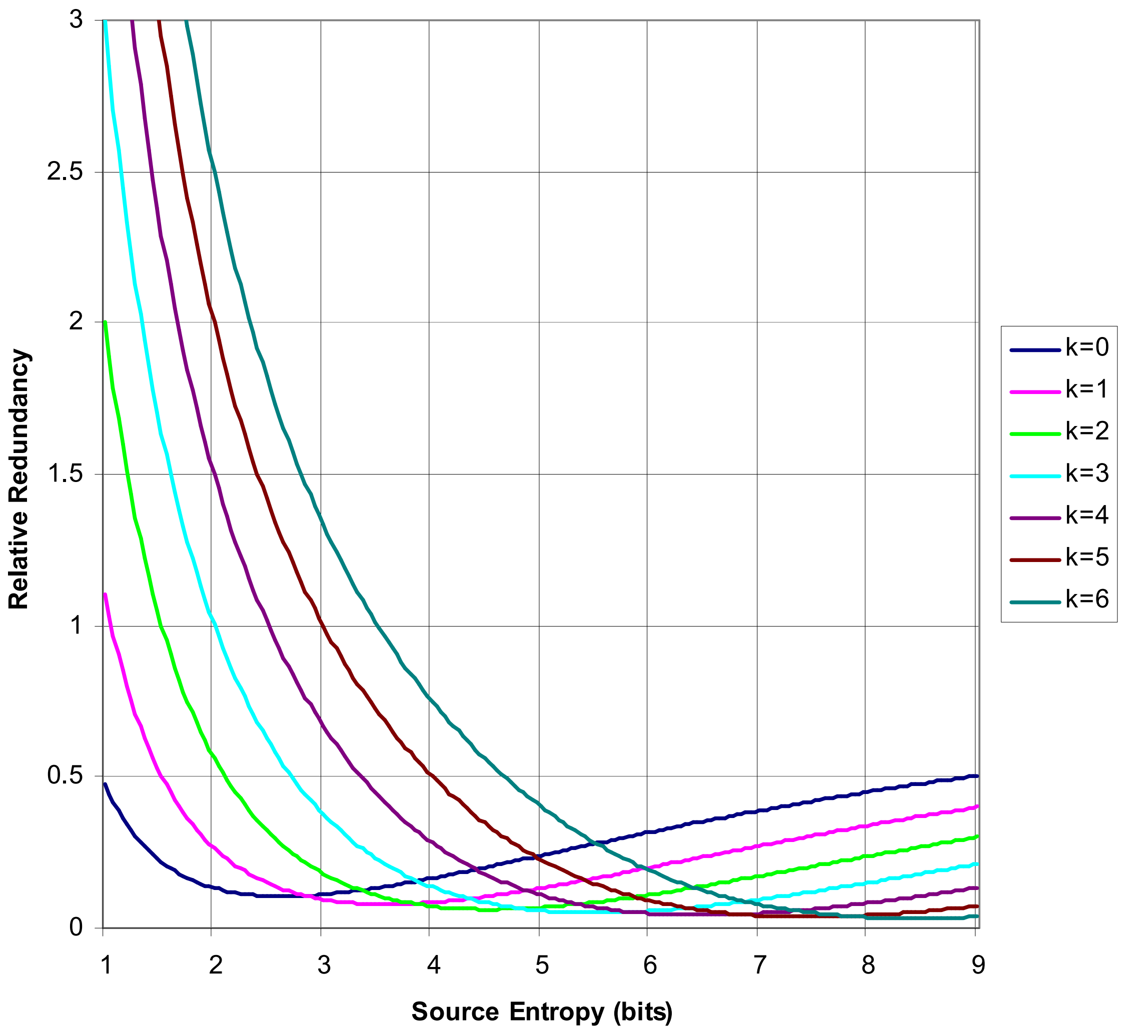

Figure 10 – Relative redundancy of resilient tree codes for sources with geometric distribution (exponential-growth, parameter w = 1).

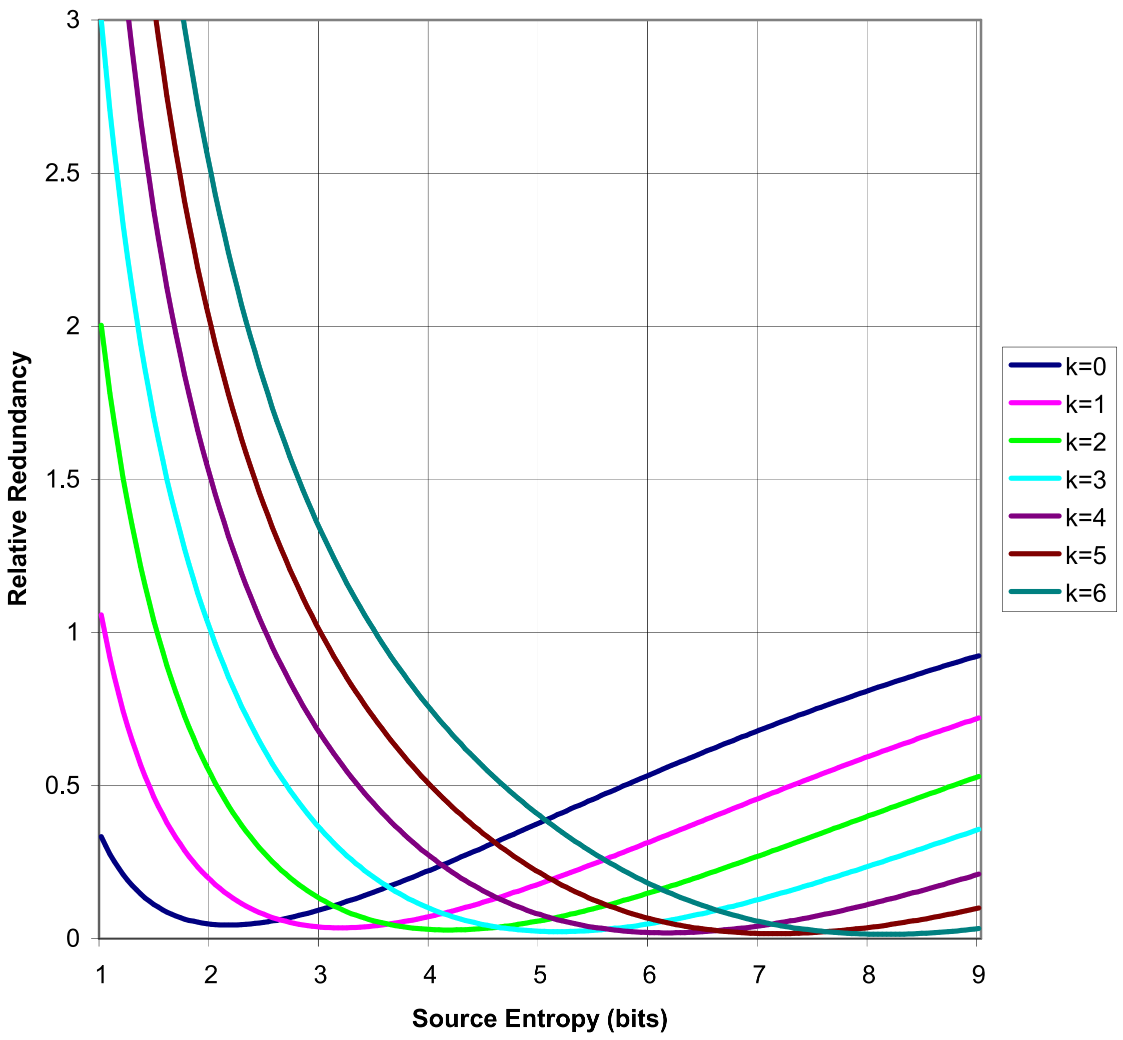

Figure 11a – Relative redundancy of resilient tree codes for sources with geometric distribution (exponential-growth, parameter w = 2).

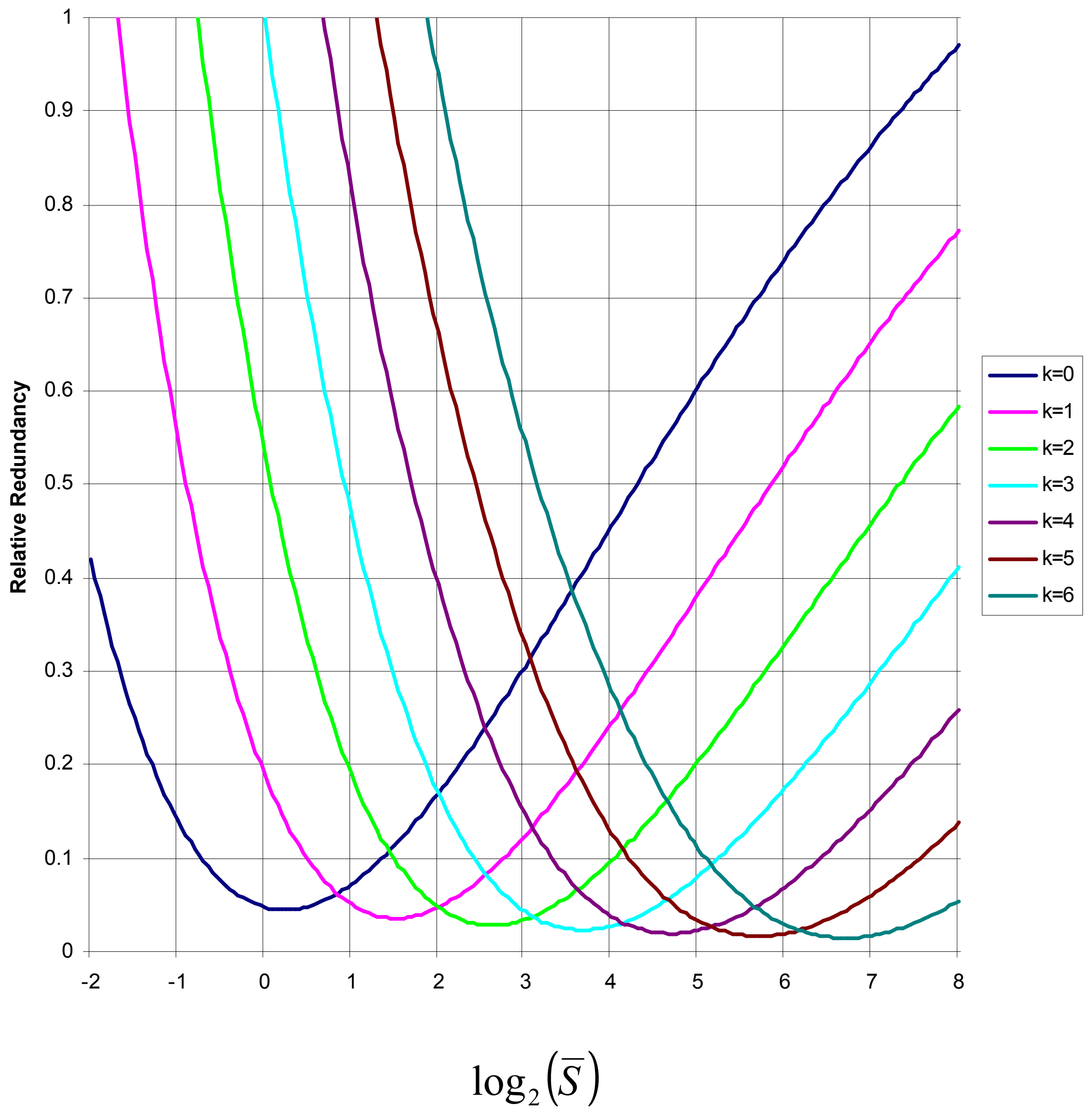

Figure 11b – Relative redundancy of resilient tree codes for sources with geometric distribution (exponential-growth, parameter w = 2).

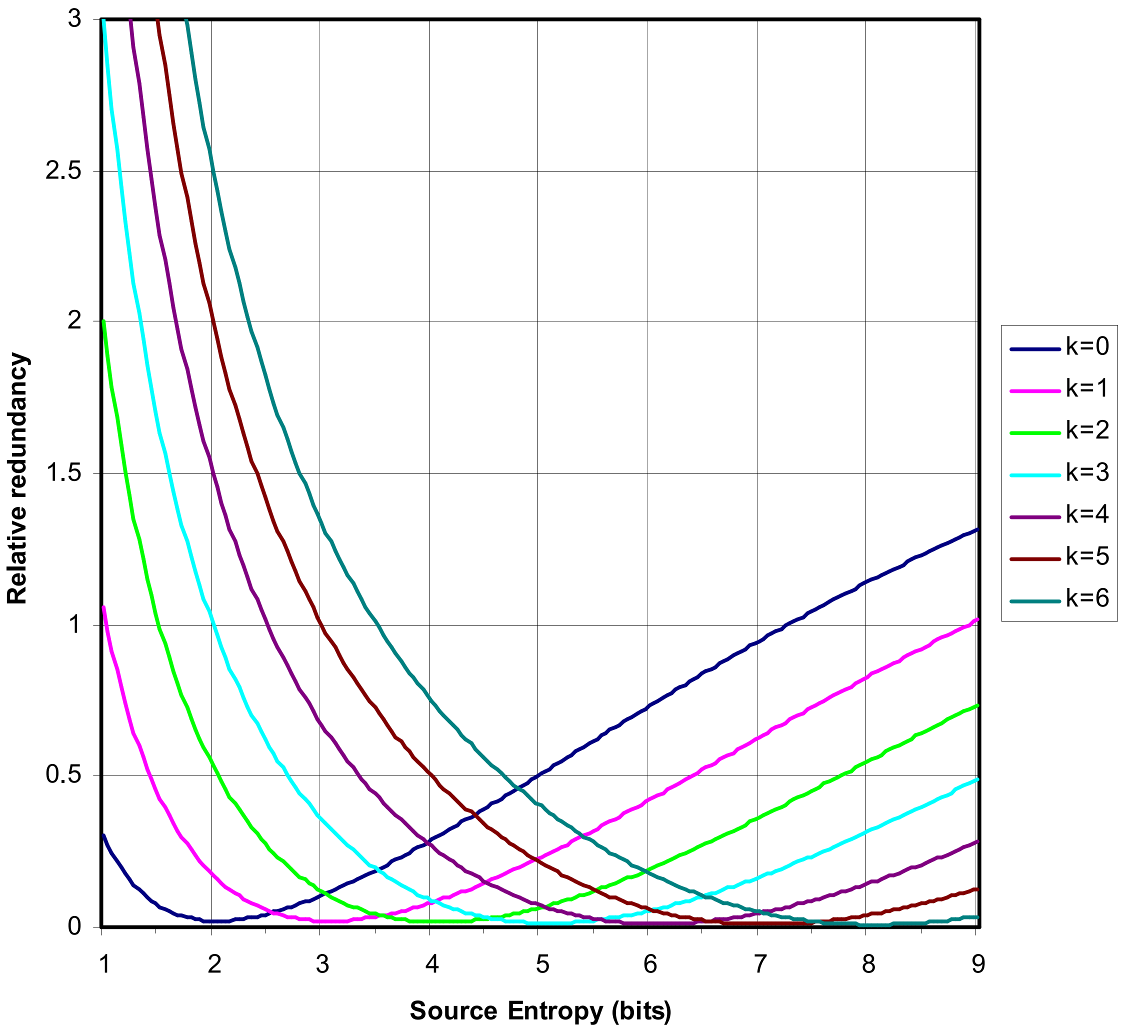

Figure 12 – Relative redundancy of resilient tree codes for sources with geometric distribution (exponential-growth, parameter w = 3).

6 Combination with Adaptive Binary Arithmetic Coding

The changes that we propose for prefix codes greatly reduce the inefficient compression that occurs when the Golomb code parameters *m* or *k* had been underestimated. There still may be some loss in compression when these parameters are overestimated. Thus, the classification should be chosen in such a way that is slightly biased towards small values of parameters *m* or *k*.

In this section we propose another solution, which is a hybrid between the approaches of estimating full probability distributions, and estimating only a code parameter, combining benefits from both. Before presenting it, we need to explain some fundamental facts about arithmetic coding.

Binary arithmetic coding is a particular form of arithmetic coding in which the source alphabet is binary, i.e., composed of only the symbols 0 and 1. Just like any set of numbers can be represented using only binary symbols (bits), several binary arithmetic encoders can be used together to code data from any data source [4,5,7].

The advantage of using binary arithmetic coders is that they can be computationally very efficient, because the binary alphabet lets us eliminate some arithmetic operations and simplify the coding process. In addition, the estimation of probabilities is simpler when we have only two symbols.

On the other hand, to code symbols from an *M*-symbol alphabet we need $M - 1$ binary coders, organized in a tree structure. When adaptive coding is used, each adaptive encoder and decoder estimates the conditional probabilities for each of the tree branches. For example, in all the code trees shown in Figures 3, 4, 7, and 8, we can have an adaptive coder assigned for each non-leaf node, and instead of outputting one bit for each transition, we can use the corresponding arithmetic code to compress the information about the transition (i.e., which branch chosen) [7].

All the problems previously mentioned about adaptive coding would apply here. For instance, if the alphabet size *M* is large, then we need large amounts of memory to store the data of all the adaptive encoders. There is also a coding speed limitation [8]. What we propose is to use the new tree structures, and use adaptive arithmetic only for the unary code. For example, Figure 13 shows a tree structure with exponential growth and the darkened nodes are those in which arithmetic coding is used. Note that in this example only four of these nodes are implemented with arithmetic coding. We did this to show that there is an infinite number of nodes in the sequence defining the unary code (top of the tree diagrams), but can choose to have only a few replaced by arithmetic coding.

The advantage is that in the new coding system the number of adaptive arithmetic coders is that we can have very complex modeling of the data, but with fast adaptation, and achieve nearly optimal compression using only a small number of adaptive arithmetic coders. Note that in Figure 13 the number of binary arithmetic coders now can be proportional to the *logarithm* of the alphabet size.

In Figure 14 we show the performance of this coding method for geometric distributions, which can be compared to Figure 4 (the axes are different, but the range is roughly the same). Note that the increase in rate caused by the inefficiency of the unary code is here completely eliminated, since the adaptive arithmetic coders can always achieve optimal compression.

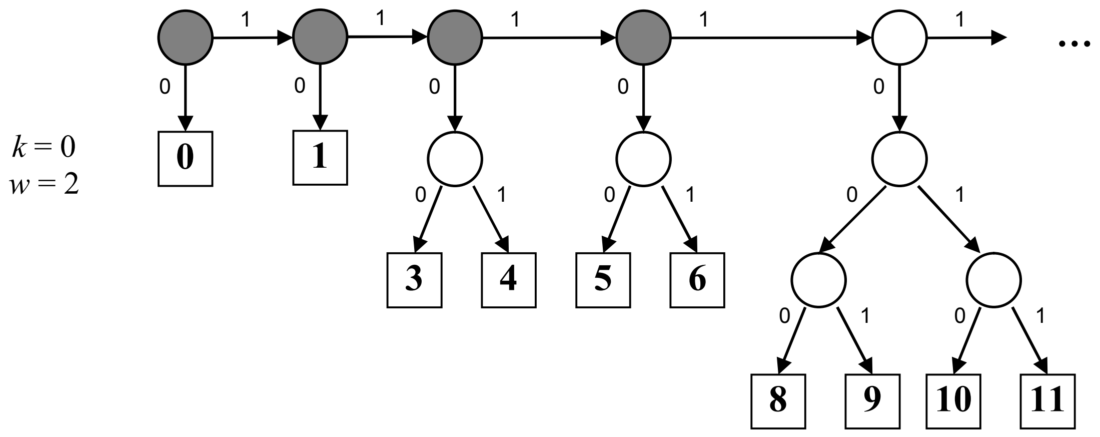

Figure 13 – Examples of resilient tree codes with exponential growth, where the darkened nodes represent adaptive binary arithmetic codes.

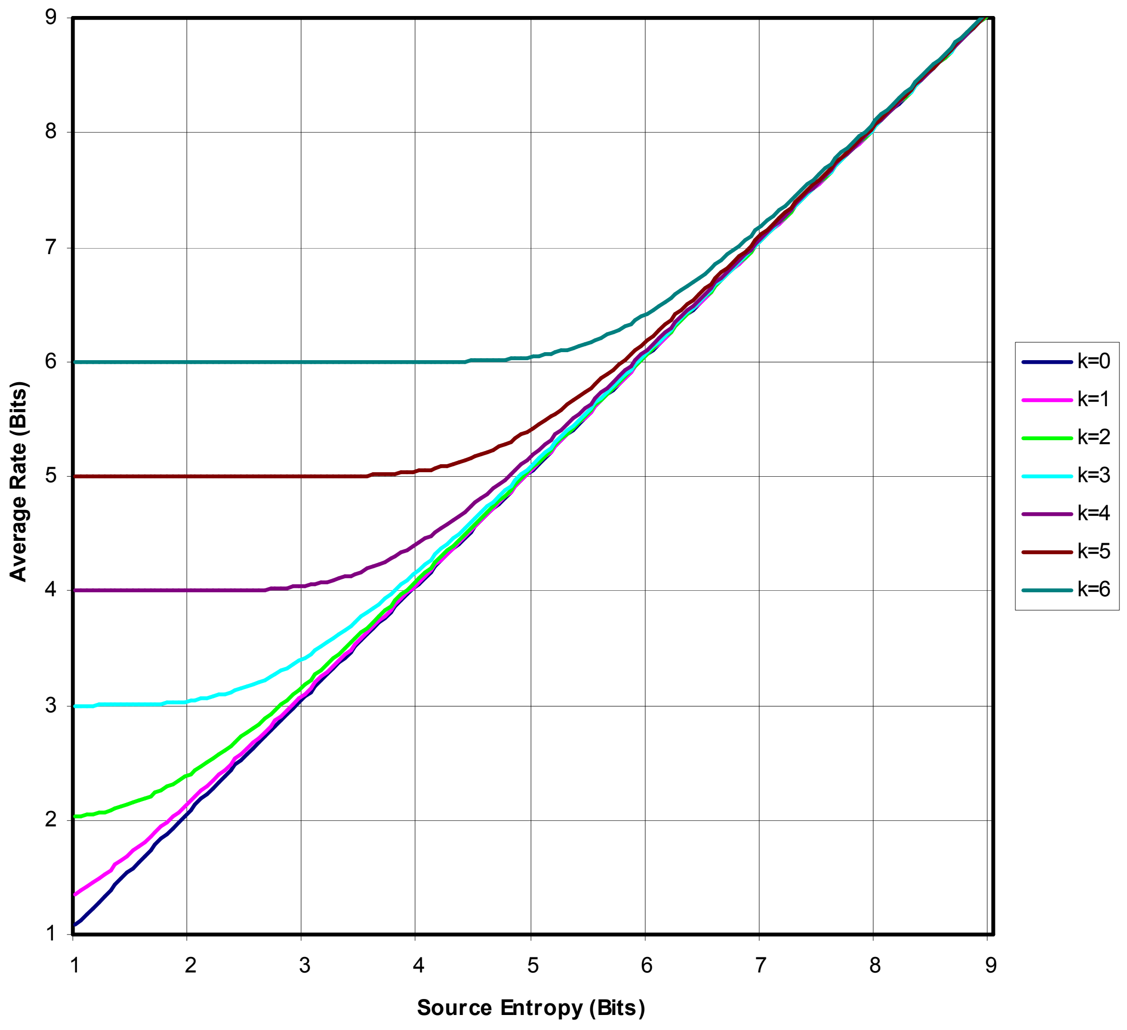

Figure 14 – Rates achieved by resilient tree codes with adaptive binary arithmetic coding for the unary part, for sources with geometric distribution (exponential-growth, parameter w = 1).

Figure 15 shows the relative redundancy, which can be compared with Figure 10. Note that for this coding scheme there is very little loss in compression when the parameter k is underestimated. On the other hand, the computational complexity increases if k is underestimated, since arithmetic coding will be used more frequently. The conclusion is that we still need good estimators for k to achieve both optimal compression and efficient coding, but now we can reduce the complexity of the estimator because the consequences of errors in estimation are much less severe.

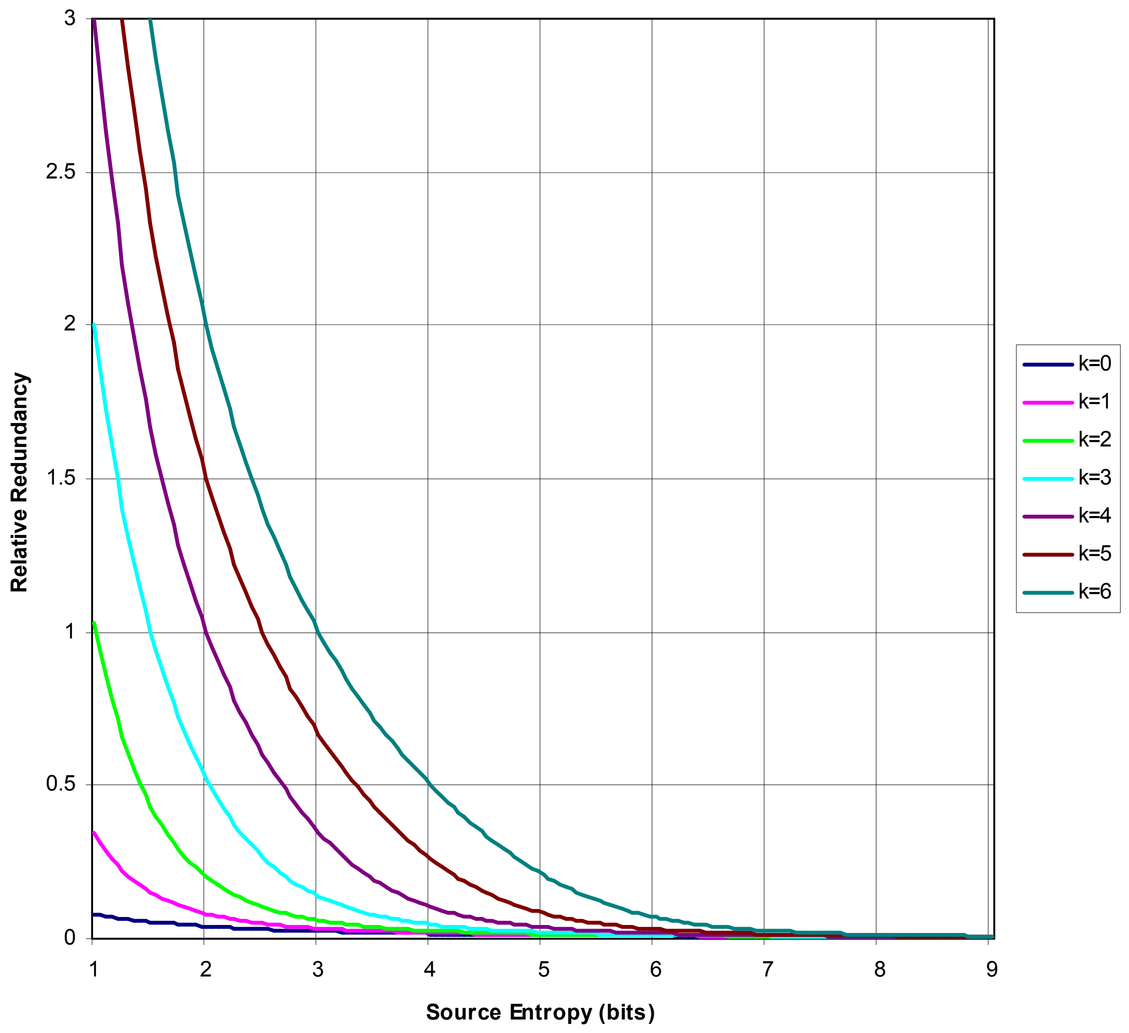

Figure 15 – Relative redundancy of resilient tree codes with arithmetic coding for the unary part, for sources with geometric distribution (exponential-growth, parameter *w* = 1).

An important question is how much loss in compression is due to the fact that some of the information is always coded with an integer number of bits (sub-trees on the lower part of Figure 13). In Figure 15 we observe this loss as the time that takes for the curves to reach the *x*-axis. In this case we can reduce this loss by increasing the value of parameter *w*. For instance, comparing Figures 15 and 16 we can see how this loss is significantly reduced when *w* is increased from 1 to 2.

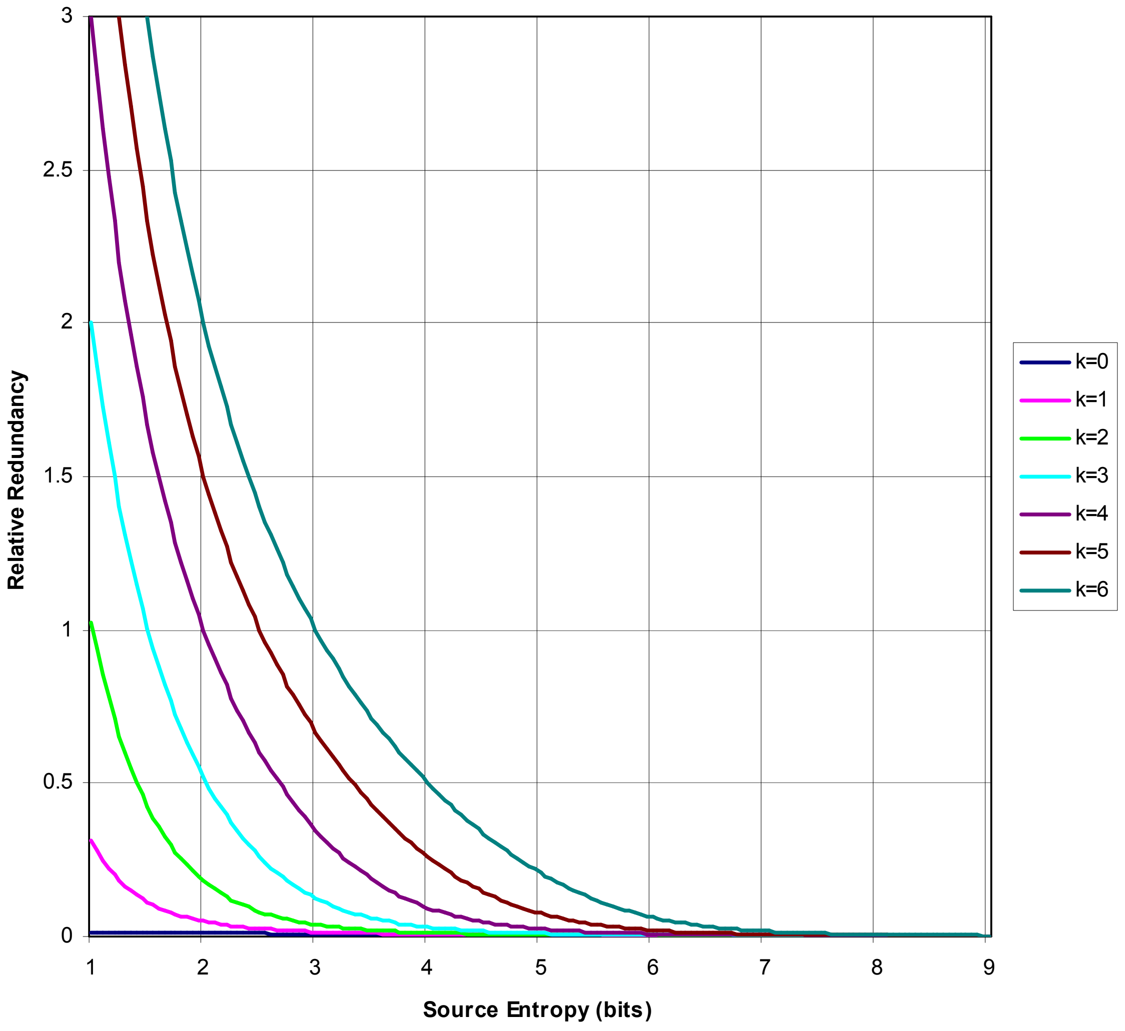

Figure 16 – Relative redundancy of resilient tree codes with arithmetic coding for the unary part, for sources with geometric distribution (exponential-growth, parameter *w* = 2).

7 Conclusion

In this report we start presenting an introduction to techniques for efficient adaptive coding. We explain that an ideal adaptive coding can be quite complex because it must reliably estimate the probability of a number of data symbols, each in number of contexts, and then create the codes for each context, and finally code the data. It is quite difficult to precisely estimate all the required data—symbols times contexts—which in can be millions of estimates. A practical alternative with much smaller complexity uses a pre-defined group of codes with special structure, and the adaptive coding process is simplified to estimating which would be the best code—among the pre-defined set—for a given symbol.

This approach became quite popular with the use of Golomb-Rice codes, since these codes are generated according to simple rules, instead of stored in tables. However, we demonstrate with simple experiments that these codes are quite sensitive to errors in the code-selection process, which are in practical applications unavoidable, due to statistical uncertainty and use for coding non-stationary sources. We propose a new family of codes that would have inferior performance if applied to stationary sources, but that are much more "resilient" to estimation error in the code selection parameters. C++ source code that exemplifies the implementation of the new codes is provided.

We propose a combination of those codes with arithmetic coding, in order to obtain nearly optimal compression, but with complexity (both memory and computation time) much lower than required for arithmetic-only coding. It is based on the fact that for these types of code, if we add a bias to the code selection to avoid overestimation, we obtain nearly optimal compression using arithmetic codes for replacing only the unary part of the prefix codes.

References

[1] S.W. Golomb, "Run-length encodings," *IEEE Trans. Information Theory,* vol. 12, pp. 388–401, July 1966.

[2] K. Sayood, *Introduction to Data Compression*, 2nd ed., Morgan Kaufmann Pubs., 2000.

[3] T.M. Cover, and J.A. Thomas, *Elements of Information Theory*, John-Wiley & Sons, 1991.

[4] G.G. Langdon, "An introduction to arithmetic coding," *IBM J. Res. Develop.,* vol. 28, no. 2, pp. 135–149, March 1984.

[5] K. Sayood, ed., *Lossless Compression Handbook*, Academic Press, 2003.

[6] G. Seroussi and M.J. Weinberger, "Efficient sequential parameter estimation for Golomb-Rice codes, with application to image compression," *Hewlett Packard Laboratories Report* HPL-96-15, Feb. 1996.

[7] A. Said, "Introduction to arithmetic coding theory and practice," *Hewlett Packard Laboratories Report* HPL-2004-76, April 2004.

[8] A. Said, "Comparative analysis of arithmetic coding computational complexity," *Hewlett Packard Laboratories Report* HPL-2004-75, April 2004.

[9] R.G. Gallager and D.C. Van Voorhis, "Optimal source codes for geometrically distributed integer alphabets," *IEEE Trans. Inform. Theory,* vol. 21, pp. 228–230, March 1975.

Appendix: C++ Implementations

Below we have a set of C++ implementations of the Golomb codes, and the new resilient tree codes. In all codes we assume that four functions to input and output individual bits to files are defined and implemented elsewhere. Their prototypes are:

```
unsigned Write_Bit(unsigned data_bit);
unsigned Write_Bits(unsigned number_of_bits, unsigned data_bits);
unsigned Read_Bit(void);
unsigned Read_Bits(unsigned number_of_bits);
```

The first function `Write_Bit`, returns the value of the bit it is saving. This helps simplifying the implementations below. We assume that the function `Write_Bits` saves the bits starting with the most-significant bits first. This enables increasing the efficiency of the decoders when parameter *m* is not a power of two, because the decoder can read $\lfloor \log_2 m \rfloor$ bits, and based on their value decide if one more bit has to be read.

A.1 Golomb Encoder and Decoder using Parameter *m*

```
void Golomb_Encoder(unsigned S, unsigned m)
{
    unsigned k = 1;
    while (m >> k) ++k;

    while (Write_Bit(S > m)) S -= m;

    if (S < (1 << k) - m)
        Write_Bits(k - 1, S);
    else
        Write_Bits(k, S);
}

unsigned Golomb_Decoder(unsigned m)
{
    unsigned S = 0;
    while (Read_Bit() == 1) S += m;

    unsigned k = 1;
    while (m >> k) ++k;

    unsigned temp = Read_Bits(k - 1);
    if (temp < (1 << k) - m)
        S += temp;
    else
        S += temp + (Read_Bit() << k);
}
```

A.2 Golomb-Rice Encoder and Decoder using Parameter *k*

```
void Golomb_Rice_Encoder(unsigned S, unsigned k)
{
    unsigned m = 1 << k;
    while (Write_Bit(S > m)) S -= m;
    Write_Bits(k, S);
}

unsigned Golomb_Rice_Decoder(unsigned k)
{
    unsigned S = 0;
    unsigned m = 1 << k;
    while (Read_Bit() == 1) S += m;
    S += Read_Bits(k);
}
```

A.3 Resilient Linear-Growth Tree Encoder and Decoder

```
void Resilient_LG_Tree_Encoder(unsigned S, unsigned m, unsigned w, unsigned d)
{
  unsigned c = 0;

  while (Write_Bit(S >= m) == 1) {
    S -= m;
    if (++c >= w) {
      c = 0;
      m += d;
    }
  }

  unsigned k = 1;
  while (m >> k) ++k;
  unsigned v = (1 << k) - m;

  if (S < v)
    Write_Bits(k - 1, S);
  else
    Write_Bits(k, v + S);
}

unsigned Resilient_LG_Tree_Decoder(unsigned m, unsigned w, unsigned d)
{
    unsigned S = 0;
    unsigned c = 0;
    while (Read_Bit() == 1) {
        S += m;
        if (++c >= w) {
            c = 0;
            m += d;
        }
    }

    unsigned k = 1;
    while (m >> k) ++k;
    unsigned v = (1 << k) - m;

    unsigned temp = Read_Bits(k - 1);
    S += temp;
    if (temp >= v)
        S += temp - v + Read_Bit();

    return S;
}
```

A.4 Resilient Exponential-Growth Tree Encoder and Decoder

```
void Resilient_EG_Tree_Encoder(unsigned S, unsigned k, unsigned w)
{
    unsigned c = 0;
    unsigned m = 1 << k;

    while (Write_Bit(S >= m) == 1) {
        S -= m;
        if (++c >= w) {
            c = 0;
            m = 1 << ++k;
        }
    }

    Write_Bits(k, S);
}

unsigned Resilient_EG_Tree_Decoder(unsigned k, unsigned w)
{
    unsigned S = 0;
    unsigned c = 0;
    unsigned m = 1 << k;

    while (Read_Bit() == 1) {
       S += m;
       if (++c >= w) {
           c = 0;
           m = 1 << ++k;
       }
    }
    S += Read_Bits(k);

    return S;
}
```