# Error-Resilient LZW Data Compression

Yonghui Wu Stefano Lonardi
Dept. Computer Science & Engineering
University of California
Riverside, CA 92521
{yonghui|stelo}@cs.ucr.edu

Wojciech Szpankowski
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
spa@cs.purdue.edu

**Abstract**

## 1 Introduction

As many practitiones know, compressed streams are very sensitive the transmission errors. Even a single error can have devastating effects and compromise all the data downstream [5]. In fact, the non-resilience of adaptive data compression has been a practical drawback of its use in many applications. Joint source-channel coding [4, 1] has emerged as a possible solution to this problem. Usually, joint source-channel coding trades source bits for channel bits or vice versa, and more than often requires some adjustments in source coding and channel coding parts.

In this paper, we deal with a very popular compression scheme, namely the so-called Lempel-Ziv-Welch (LZW). LZW is a lossless data compression algorithm developed by T. Welch in 1984 [8] as an improved version of the LZ78 dictionary coding algorithm developed by A. Lempel and J. Ziv [9]. The method became moderately widely-used when the program `compress` became more or less the standard compression utility in Unix systems circa by the year of 1986. In 1987, the algorithm was adopted as part of the GIF image format, and has since been very widely used. LZW is used in the V.42bis modem standard [7] and can also optionally be used in TIFF images and PDF formatted documents.

Here focus on the problem of adding error-resiliency to the LZW compression scheme. Compared to our previous work on LZ-77 [1], extracting from LZW/LZ-78 the extra redundancy bits needed to store the error correcting parity bits turned out to be significantly trickier. The constrains that we had in the design of the new scheme were mainly two. First, we wanted to maintain the backward compatibility with the original LZW, that is, we wanted a file compressed with LZW and augmented for error-resiliency to be decodable by the original LZW. Backward-compatibility is very desirable property because it makes it possible to deploy the new scheme gradually over the existing one, without disrupting service. Second, we wanted the compression ratio not to be affected too much by the embedding of the extra parity bits for the detection and correction of errors.

We were able to achieve both objectives by relaxing (i.e., shortening) a few selected LZW phrases in the compressed stream, so that the pattern of shorter phrases would encode the extra

bits. Since we are not shortening too many phrases the compression is not affected much (see Experimental results). In fact, according to our experimental results, sometimes the sum of the

## 2 Basic concepts

Let $T$ be a text of length $n$ over a finite alphabet $\mathcal{A}$, and let $T'$ be its corresponding LZW-compressed stream. We write $T_{[i]}$ to indicate the $i^{th}$ symbol of the text. We use $T_{[i,j]}$ shorthand for $T_{[i]}T_{[i+1]}...T_{[j]}$, where $1 \leq i \leq j \leq n$, with the convention that $T_{[i,i]} = T_{[i]}$. Substrings in the form $T_{[1,i]}$ correspond to the *prefixes* of $T$, and substrings in the form $T_{[i,n]}$ to the *suffixes* of $T$. Given two strings $y$ and $w$, $y + w$ is the string obtained by concatenating $y$ with $w$.

We use $\mathcal{D} = \{D_1, D_2, \ldots\}$ to denote the LZW dictionary used during the encoding and decoding processes. By construction of the LZW dictionary we have that $D_i = T_{[j,k]}$ for some $j$ and $k$. The length $|D_i|$ of a phrase $D_i$ is the number of symbols it contains, i.e., $|D_i| = |T_{[j,k]}| = k - j + 1$.

For completeness of presentation we briefly review the original LZW scheme [8, 9]. The algorithm parses the text *online* left to right into phrases, where each phrase is the longest matching phrase seen previously plus one extra symbol. Each new phrase is added to the dictionary $\mathcal{D}$ of phrases, which is first initialized to include every single symbol in the input alphabet. The index of the longest matching phrase is added to the output $T'$ as a new codeword, whereas the extra symbol, i.e., the last symbol of the current phrase, becomes the first symbol of the next phrase.

To decode the compressed text $T'$, the decoder first fills the dictionary $\mathcal{D}$ with all the symbols in the input alphabet. It then reads the codewords one by one from the compressed text $T'$. Every time the decoder reads a new codeword, it looks up the dictionary for the corresponding phrase. The string identified by the codeword is added to the output, while a new phrase, which is constructed by appending the first symbol of the current codeword to the previous codeword, is added to the dictionary $\mathcal{D}$.

## 3 Extracting bits from the LZW stream

Due to the greediness inherent to the LZW algorithm, at any point of the encoding process there is always only one way of producing the next phrase, and hence, every phrase in the dictionary $\mathcal{D}$ is unique. The greediness prevents us from embedding directly extra bits into the compressed data stream. A simple solution to this problem is to relax the greediness on some of the phrases (i.e., by making them a little shorter) in such a way to encode the extra bits. Relaxing the length of *too many* phrases will, however, degrade the compression performance considerably. Care must be taken to select the set of phrases to shorten that will allow the necessary "extra space" to store the bits for error detection and correction. Also, we must ensure that the new compressed stream can be decompressed by the original algorithm. Note that by relaxing the greediness some entries in the dictionary $\mathcal{D}$ will have multiple codewords associated with them. A somewhat similar approach was taken in [6], where the author analyze a scheme where the entries in the dictionary to have multiplicity up to a constant $b$.

Figure 1 show an example. On the left, the text $T$=`aaaaaab` is compressed with the greedy LZW algorithm. The corresponding compressed data stream is $T'$=`0231`. By the end of the encoding process, the dictionary contains five phrases, namely $D_0$=`a`, $D_1$=`b`, $D_2$=`aa`, $D_3$=`aaa`,

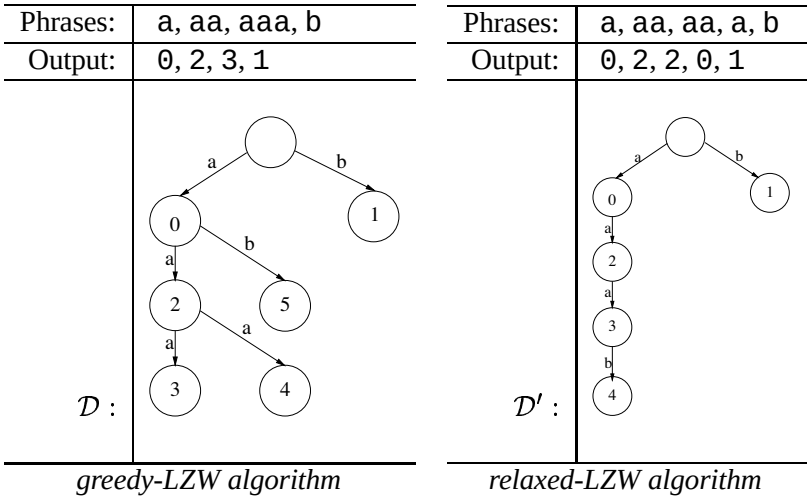


Figure 1: Greedy-LZW parsing vs. relaxed-LZW parsing on the input $T$=aaaaaab

and $D_4$=aaab. Phrases $D_0$ and $D_1$ are from the input alphabet $\mathcal{A}$, whereas $D_2$, $D_3$, and $D_4$, are constructed greedily as the text is parsed from left to right. If instead of generating the next phrase in a greedy manner all the time, we reduce the length of $D_4$ by one, we will get the result as shown in the right part of the figure. This time, the dictionary contains 6 phrases and the output, 02201, is one codeword longer than that in the greedy case. The two phrases, $D'_3$ and $D'_4$, are identical, which is as expected. The dictionary $\mathcal{D}$ and $\mathcal{D}'$ are both shown as tries in the figure.

The crucial question is whether this modification in the encoder will produce a compressed file that can still be decompressed by the original LZW algorithm. The answer to this question depends on the data structure used to represent the dictionary. Although in the literature, the dictionary is always represented as a trie, all the implementation we checked use a fixed-size hash table (typically with 4096 entries). Thus, the decoder is not affected by the multiplicities in the dictionary entries. All it does is to refer to the existing phrases by their indices, whether there are multiple such phrases or not, and concatenates two strings to produce the next phrase. Because of this, our relaxed-LZW scheme is backward-compatible with the greedy-LZW. We verified the backward compatibility of our scheme on various software that support the LZW scheme. For example, for GIF images, we tested MS Paint, MS Internet Explorer and Mozilla Firefox. We also tested Unix Compress and Winzip. All these latter software were able to decompress a relaxed-LZW stream. We expect PDF and TIFF files to be capable of handling our scheme as well.

Once the backward compatibility have been assessed, we need to take the compression performance into consideration. A detailed description on the relaxed-LZW scheme is in order. We call the LZW phrases that will become shorter because of the relaxation, *non-greedy* phrases.

Let $M$ denote the message over the binary alphabet that is going to be embedded into the compressed text $T'$. The capacity of the "embedding channel" depends on two integer parameters $K$ and $L$. The integer $K$ specifies the number of bits of the message that are embedded in the interval between two consecutive non-greedy phrases, whereas the $L$ parameter specifies the number of bits that is embedded in the length of each non-greedy phrase. For example, when $K = 1$ we ...

The message $M$ to be embedded is first logically divided into consecutive blocks of $K + L$ bits. Every block is further divided into two sub-blocks of $K$ and $L$ bits each. The contents of the

```
MESSAGE_EMBEDDING_LZW_ENCODER(T, M, K, L)
1.  T', i ← ∅, 0
2.  k ← get next K bits from M
3.  if (k = 0) then k ← 2^K
4.  while (have not finished encoding yet)
5.      phrase ← next phrase as according to the standard LZW algorithm
6.      if (|phrase| > 2^L) then
7.          i ← i + 1
8.          if (i = k) then
9.              i ← 0
10.             l ← get next L bits from M
11.             if (l = 0) then l ← 2^L
12.             phrase ← reduce the length of phrase by l symbols
13.             k ← get next K bits from M
14.             if (k = 0) then k ← 2^K
15.     T' ← T' + phrase
16. return T'
```

Figure 2: The sketch of the encoder capable of embedding a message $M$ while compressing $T$ into a LZW stream. $K$ and $L$ are two integer parameters of the algorithm (see text)

two sub-blocks are interpreted as positive integers, and for clarity purposes, we denote them by $k$ and $l$ respectively. Note that[1] $k \in [0, 2^K - 1]$ and $l \in [0, 2^L - 1]$. The message $M$ is embedded into $T'$ one block a time while compressing $T$ according to LZW, as follows. We initialize a counter $i$ to 0, and generate new phrases greedily as per the greedy LZW algorithm. Every time a new phrase is generated, its length is compared to the value $2^L$ (note that $2^L$ is the maximum value for $l$, see footnote). If the length of the phrase is greater we increment the counter by 1. As soon as the counter equals the value $k$ we relax the length of that phrase by $l$ symbols. Then, the counter is reset to 0, and a new cycle begins (see Figure 2)

The decoding process is rather straightforward. As codewords are read from $T'$, phrases are being reconstructed. For each phrase the decoder determines whether the phrase is greedy or non-greedy. If the phrase is non-greedy phrase, a block of message is recovered according to the rules we followed to embed it. At the same time, the original text is also rebuilt as per the generic LZW algorithm. Note that in order to determine whether or not a phrase is non-greedy, we need to look ahead several phrases. This can be done by employing some sort of look-ahead buffer. A sketch of the decoder is illustrated in Figure 3.

As a final remark, we want to note that in the strategy described above we have not exploited the multiplicities in the dictionary to embed even more bits of the message. When the relaxed-LZW algorithm looks for a longest prefix of text to be compressed that is contained the dictionary $\mathcal{D}$, there might be multiple such longest phrases in the dictionary, due to the fact that we have reduced the lengths of some of the previous phrases. If that is the case, we can embed "free of charge" another $\lfloor \log_2 q \rfloor$ where $q$ is the multiplicity of the longest phrase. A similar idea is exploited in [1, 2] to embed extra bits in LZ-77 streams.

[1] $k = 0$ and $l = 0$ are treated as a special case, by mapping them to $2^K$ and $2^L$, respectively.

MESSAGE_EMBEDDING_LZW_DECODER$(T', K, L)$

1. $T, M, buffer, i \leftarrow \emptyset, \emptyset, \emptyset, 0$
2. **while** (have not finished decoding yet)
3. $\quad phrase1 \leftarrow$ decode the next phrase as according to the standard LZW decoder
4. $\quad T \leftarrow T + phrase1$
5. $\quad$ Fill $buffer$
6. $\quad phrase2 \leftarrow$ encoder the text in $buffer$ as according to the standard LZW encoder
7. $\quad$ **if** $(|phrase2| > 2^L)$
8. $\quad\quad i \leftarrow i + 1$
9. $\quad\quad$ **if** $(|phrase1| < |phrase2|)$
10. $\quad\quad\quad M \leftarrow M + (i \& 2^K) + ((|phrase2| - |phrase1|) \& 2^L)$
11. $\quad\quad\quad i \leftarrow 0$
12. **return** $(T, M)$

Figure 3: The sketch of the decoder capable of recovering the embedded message $M$ from a LZW stream $T'$. $K$ and $L$ are two integer parameters of the algorithm (see text)

# 4 Selection of parameters $K$ and $L$

Embedding extra information into the LZW data stream is clearly not free of charge. With additional bits embedded, the size of the new LZW stream is usually larger than that of the original one, as is illustrated by the example in Figure 1. However, by choosing the two parameters $K$ and $L$ judiciously, the size will not increase by too much. In fact, we will show in the Section 7 that when $K$ is 5 or higher, the size of the new LZW file is typically less than the sum of the sizes of the original file and the message file.

The compression tend to degrade as $K$ decreases and $L$ increases, but at the same time the number of bits embedded in the message will increase. In the error resilient application it is crucial to determine the best tradeoff. Once the file is compressed with the original greedy-LZW, the stream is broken into blocks (see Section 6) and the total number of parity bits will be computed. In this section, we discuss how to choose $K$ and $L$ so that we can estimate the number of bits that can be embedded in $T'$. The aim is to create enough extra bits for the length of the message to be embedded, i.e., all the parity bits of the error-correcting code, but not much more so that the compression will suffer.

In our analysis, we assume that during the embedding of the message, the lengths of the phrases are always greater than $2^L$, which of course is not true at the beginning of the file. However, as long as the text is long enough and as new phrases are being generated and inserted into the dictionary, the assumption is likely to be satisfied. For simplicity, we assume the message $M$ to be embedded is generated by an i.i.d. model with `0` and `1` having equal probabilities. Then, on average we will get a non-greedy phrase every $(2^K+1)/2$ phrases. On average, the length of the non-greedy phrase will be reduced by $(2^L + 1)/2$ symbols. For simplicity in our exposition, we set $S_1 = (2^K + 1)/2$ and $S_2 = (2^L + 1)/2$.

Let assume that $|T| = n$ and $|M| = m$. Let us call $T_1$ the portion of $T$ that is encoded by greedy phrases, and call $T_2$ the portion of $T$ encoded by non-greedy phrases. Intuitively, if we "zip" $T_1$ and $T_2$ at the points where $T$ is broken into phrases, we get back $T$. Let the sizes of $T_1$ and $T_2$ be $n_1$ and $n_2$ respectively. Clearly, $n = n_1 + n_2$.

The set of unique phrases in the dictionary is determined by the greedy phrases (in $T_1$). The number of the unique phrases, $P'$, is then approximately equal to $\frac{n_1\mathcal{H}}{\log n_1}$, where $\mathcal{H}$ is the entropy of $T_1$. The average length of the greedy phrases, $l'$, will be roughly $\frac{n_1}{p'} = \frac{\log(n_1)}{\mathcal{H}}$. The number of blocks of message, $B$, that is embedded in the text will be $\frac{P'}{S_1}$, and hence we have $m = B(K+L) = \frac{P'}{S_1}(K+L)$. Let $l''$ be the average length of the non greedy phrases. Then we have $l'' = l' - S_2$, and $n_2 = Bl" = \frac{P'}{S_1}(\frac{n_1}{P'} - S_2) = \frac{n_1}{S_1} - \frac{n_1\mathcal{H}S_2}{\log(n_1)*S_1}$. The number of non-greedy phrases in the dictionary is equal to the number of blocks of messages embedded. The total number of phrases, $P$, generated by our algorithm is the sum of that of greedy phrases and of the non-greedy phrases, which is equal to $P' + B$. To sum up, we have the following equalities:

$$\begin{aligned}
S_1 &= \frac{2^K+1}{2}, S_2 = \frac{2^L+1}{2} \\
n_2 &= B(l' - S_2) = \frac{P'}{S_1}(\frac{n_1}{P'} - S_2) = \frac{n_1}{S_1} - \frac{n_1\mathcal{H}S_2}{\log(n_1)S_1} \\
n &= n_1 + n_2 = n_1 + \frac{n_1}{S_1} - \frac{n_1\mathcal{H}S_2}{log(n_1)S_1} \\
m &= B(K+L) = \frac{P'}{S_1}(S_1+S_2) = \frac{n_1\mathcal{H}}{S_1\log n'}(S_1+S_2)
\end{aligned}$$

From the above set of equations, $n_1$ can be determined given $n$ and $\mathcal{H}$, so will be $B$, $P$ and $m$. The capacity of our message-embedding channel can be represented as the ratio of $m$ over $n$. The performance of our algorithm, in terms of compression ratio, can be determined by comparing the total number of phrases, $P$, that is generated by our algorithm to the number of phrases, $\frac{(n+m)\mathcal{H}}{log(n+m)}$, that would be produced if we were to compress the concatenation of the text and the message $T + M$ simply by the generic LZW algorithm.

# 5 Asymptotic analysis of the redundancy

In this section we are concerned with the analysis of the redundancy of the new scheme when the input is large. In particular we are interested in comparing the redundancy of the relaxed-LZW scheme with the original LZW/LZ-78. Here, we follow the notation from [3]. On average our relaxed-LZW algorithm decreases the number of manipulated symbols by $x_n L/K$ where $x_n$ is the yet unknown (average) number of phrases when a string of length $n$ is compressed.

Let, as in [3], define

$$\mu(x) = \frac{x}{h}\log x - \frac{A}{h}x - \frac{x}{K}L + O\left(\frac{\log x}{h}\right) \tag{1}$$

where

$$A = 1 - \gamma - \frac{1}{2h}h_2 + \alpha - \delta_0(n)$$

and where $h = -p\log p - q\log q > 0$ is the entropy, $\gamma = 0.577\ldots$ is the Euler constant, $h_2 = p\log^2 p + q\log^2 q$, and

$$\alpha = -\sum_{k=1}^{\infty} \frac{p^{k+1}\log p + q^{k+1}\log q}{1 - p^{k+1} - q^{k+1}}\,. \tag{2}$$

The function $\delta_0(x)$ is a fluctuating function with mean zero and a small amplitude for $\log p/\log q$ rational (e.g., the amplitude of $\delta_0(x)$ is smaller than $10^{-6}$ for the unbiased case, where $p = q = 0.5$), and $\lim_{x\to\infty}\delta_0(x) = 0$ otherwise.

Define now $x_n$ as

$$\mu(x_n) = n.$$

Then, as in [3], the average number of phrases of our algorithm is $E[M_n] \sim x_n$. Observe that the code length $L_n$ of our algorithm is

$$L_n = x_n(1-1/K)[\log(x_n(1-1/K)) + 1].$$

Thus, the relative average redundancy $r_n$ becomes

$$\begin{aligned} r_n &= \frac{x_n(1-1/K)[\log(x_n(1-1/K)) + 1] - nh}{n} \\ &= h\frac{1 + A + Lh/K}{\log n}. \end{aligned} \tag{3}$$

Comparing it to the regular LZW/LZ'78, the redundancy of relaxed-LZW is increased by $Lh/(K\log n)$.

# 6 Error resilient LZW

# 7 Experimental Results

In order to validate our theoretical studies and test the correctness of our scheme, we implemented our algorithm and tested it on GIF files, which, as is well-known, is compressed by LZW algorithm. We downloaded from Internet several standard pictures for digital image processing, and tested our algorithm with them. Table 1 shows our experimental results with $K$ and $L$ being set to 5 and 0 respectively. The "Message Length (bytes)" column indicates how much random information we can embed into the original GIF file. Column "Cmprssn Loss" is defined to be the size of the new GIF file minus the sum of the size of the original GIF file plus the size of the random message. If this number is negative, it means that overall we are gaining more compression than the standard LZW algorithm. As we can see from the table, `airplane.gif` and `couple.gif` are better than the rest of the GIF files in terms of compression loss, and `baboon.gif` is the worst among all of them. This is because the compression ratio for `airplane.gif` and `couple.gif` (5.77 and 4.88 respectively) is much higher than that for `baboon.gif` (2.49), and reducing the lengths of phrases will not have as dramatic negative impact on the file size for GIF files with high compression ratios than for those with low compression ratios. Column "channel capacity" is defined to be the ratio of the size of the random message over the size of the original GIF

| | Original GIF file | | | | | Message embedded GIF file | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Width | Height | BPP | Size (bytes) | Average LZW Phrase Length | Size (bytes) | Mssg Length (bytes) | Average LZW Phrase Length | Cmprssn Loss | Channel Capacity |
| airplane.gif | 512 | 512 | 8 | 64908 | 5.77 | 66468 | 1706 | 5.63 | -146 | 0.026283 |
| baboon.gif | 512 | 512 | 8 | 149414 | 2.49 | 151804 | 2169 | 2.45 | 221 | 0.014516 |
| couple.gif | 256 | 256 | 8 | 19604 | 4.88 | 20088 | 505 | 4.77 | -21 | 0.025760 |
| girl.gif | 256 | 256 | 8 | 23573 | 4.04 | 24127 | 566 | 3.94 | -12 | 0.024010 |
| lena.gif | 512 | 512 | 8 | 96373 | 3.87 | 98770 | 2396 | 3.78 | 1 | 0.024861 |
| peppers.gif | 512 | 512 | 8 | 105262 | 3.54 | 107792 | 2372 | 3.46 | 158 | 0.022534 |

Table 1: Comparing the size of some GIF files compressed with the standard LZW versus the message embedding LZW scheme $(K = 5, L = 1)$

file. We notice that the variation of "channel capacity" among the files are quite small, except `baboon.gif`, which again we believe is due to its relatively poor compression ratio.

Table 2 shows our experimental results with various parameter settings. Clearly, the channel capacity decreases as the parameter $K$ increases. However, the relationship between the channel capacity and the parameter $L$ is not as clear. To our observations, when the parameter $K$ is small, say less than 4, increasing the parameter $L$ from 0 to 1 usually increases the channel capacity as well. However, going beyond 1 neither helps channel capacity nor does it improve the image compression ratio.

# References


[1] LONARDI, S., AND SZPANKOWSKI, W. Joint source-channel LZ'77 coding. In *IEEE Data Compression Conference, DCC* (Snowbird, Utah, March 2003), J. A. Storer and M. Cohn, Eds., IEEE Computer Society TCC, pp. 273–283.

[2] LONARDI, S., SZPANKOWSKI, W., AND WARD, M. Error resilient LZ'77 and its analysis. In *IEEE International Symposium on Information Theory (ISIT'04)* (Chicago, IL, June 2004), p. 56.

[3] LOUCHARD, G., AND SZPANKOWSKI, W. On the average redundancy rate of the Lempel-Ziv code. *IEEE Trans. Inf. Theory 43* (1997), 2–8.

[4] SAYOOD, K., OTU, H., AND DEMIR, N. Joint source/channel coding for variable length codes. *IEEE Trans. Inf. Theory 48* (2000), 787–794.

[5] STORER, J. A., AND REIF, J. H. Error-resilient optimal data compression. *SIAM Journal on Computing 26*, 4 (1997), 934–949.

[6] SZPANKOSWKI, W., AND KNESSL, C. A note on the asymptotic behavior of the height in $b$-tries for $b$ large. *Electronic J. of Combinatorics 7* (2000), R39.

[7] THOMBORSON, C. The V.42bis standard for data-compressing modems. *IEEE Micro 12*, 5 (Oct. 1992), 41–53.

| | Mssg Length (bytes) | Average LZW Phrase Length | Cmprssn Loss | Channel Capacity | Mssg Length (bytes) | Average LZW Phrase Length | Cmprssn Loss | Channel Capacity |
|---|---|---|---|---|---|---|---|---|
| | $K=1, L=0$ | | | | $K=1, L=1$ | | | |
| airplane.gif | 5295 | 3.91 | 25058 | 0.081577 | 9122 | 3.97 | 20001 | 0.14053737 |
| baboon.gif | 10973 | 1.66 | 62620 | 0.07344023 | 10768 | 1.97 | 28233 | 0.07206821 |
| couple.gif | 1509 | 3.38 | 7035 | 0.07697408 | 2650 | 3.39 | 5798 | 0.13517649 |
| girl.gif | 1823 | 2.76 | 8798 | 0.07733423 | 2978 | 2.85 | 6654 | 0.12633097 |
| lena.gif | 8003 | 2.53 | 42706 | 0.08304193 | 12624 | 2.65 | 31451 | 0.13099104 |
| peppers.gif | 8392 | 2.34 | 45190 | 0.07972487 | 12278 | 2.51 | 30728 | 0.11664228 |
| | $K=2, L=0$ | | | | $K=2, L=1$ | | | |
| airplane.gif | 5227 | 4.73 | 8759 | 0.08052936 | 6799 | 4.75 | 6950 | 0.10474825 |
| baboon.gif | 10778 | 2.02 | 23867 | 0.07213514 | 8365 | 2.2 | 11363 | 0.05598538 |
| couple.gif | 1524 | 4 | 2638 | 0.07773923 | 1979 | 4.01 | 2134 | 0.10094878 |
| girl.gif | 1794 | 3.34 | 3023 | 0.07610401 | 2207 | 3.4 | 2202 | 0.09362406 |
| lena.gif | 7776 | 3.12 | 15101 | 0.08068649 | 9337 | 3.18 | 11384 | 0.09688398 |
| peppers.gif | 8167 | 2.87 | 16189 | 0.07758735 | 9119 | 2.97 | 11055 | 0.08663145 |
| | $K=3, L=0$ | | | | $K=3, L=1$ | | | |
| airplane.gif | 3947 | 5.23 | 2720 | 0.06080914 | 4542 | 5.24 | 2035 | 0.06997596 |
| baboon.gif | 8100 | 2.23 | 8947 | 0.05421178 | 5718 | 2.33 | 4477 | 0.0382695 |
| couple.gif | 1160 | 4.45 | 779 | 0.05917159 | 1289 | 4.48 | 521 | 0.06575188 |
| girl.gif | 1356 | 3.67 | 995 | 0.05752343 | 1468 | 3.69 | 690 | 0.06227463 |
| lena.gif | 5813 | 3.47 | 5311 | 0.06031772 | 6205 | 3.49 | 4031 | 0.06438525 |
| peppers.gif | 6153 | 3.17 | 5934 | 0.05845414 | 6185 | 3.23 | 3967 | 0.05875814 |
| | $K=4, L=0$ | | | | $K=4, L=1$ | | | |
| airplane.gif | 2656 | 5.48 | 623 | 0.04091945 | 2819 | 5.49 | 366 | 0.0434307 |
| baboon.gif | 5432 | 2.36 | 3033 | 0.03635536 | 3698 | 2.4 | 1482 | 0.02475002 |
| couple.gif | 778 | 4.67 | 140 | 0.03968577 | 828 | 4.67 | 93 | 0.04223627 |
| girl.gif | 940 | 3.83 | 282 | 0.03987612 | 952 | 3.84 | 201 | 0.04038518 |
| lena.gif | 3873 | 3.66 | 1555 | 0.0401876 | 3941 | 3.67 | 1129 | 0.04089319 |
| peppers.gif | 4071 | 3.35 | 1718 | 0.03867492 | 3917 | 3.37 | 1176 | 0.0372119 |
| | $K=5, L=0$ | | | | $K=5, L=1$ | | | |
| airplane.gif | 1663 | 5.63 | -90 | 0.02562087 | 1706 | 5.63 | -146 | 0.02628335 |
| baboon.gif | 3413 | 2.42 | 756 | 0.02284257 | 2169 | 2.45 | 221 | 0.01451671 |
| couple.gif | 479 | 4.77 | -27 | 0.02443378 | 505 | 4.77 | -21 | 0.02576004 |
| girl.gif | 582 | 3.92 | 63 | 0.02468926 | 566 | 3.94 | -12 | 0.02401052 |
| lena.gif | 2441 | 3.76 | 276 | 0.02532867 | 2396 | 3.78 | 1 | 0.02486173 |
| peppers.gif | 2566 | 3.45 | 357 | 0.02437726 | 2372 | 3.46 | 158 | 0.02253424 |
| | $K=6, L=0$ | | | | $K=6, L=1$ | | | |
| airplane.gif | 987 | 5.72 | -300 | 0.01520613 | 982 | 5.71 | -181 | 0.0151291 |
| baboon.gif | 2039 | 2.46 | -130 | 0.01364664 | 1305 | 2.47 | -98 | 0.00873412 |
| couple.gif | 288 | 4.81 | -14 | 0.01469087 | 286 | 4.83 | -64 | 0.01458885 |
| girl.gif | 329 | 3.99 | -37 | 0.01395664 | 311 | 3.98 | -4 | 0.01319305 |
| lena.gif | 1478 | 3.82 | -236 | 0.01533624 | 1390 | 3.83 | -260 | 0.01442312 |
| peppers.gif | 1545 | 3.49 | -84 | 0.01467766 | 1376 | 3.49 | 25 | 0.01307214 |
| | $K=7, L=0$ | | | | $K=7, L=1$ | | | |
| airplane.gif | 591 | 5.75 | -303 | 0.00910519 | 583 | 5.75 | -294 | 0.00898194 |
| baboon.gif | 1184 | 2.47 | -127 | 0.00792429 | 782 | 2.48 | -102 | 0.00523377 |
| couple.gif | 182 | 4.84 | -38 | 0.00928381 | 175 | 4.85 | -37 | 0.00892674 |
| girl.gif | 193 | 4.01 | -22 | 0.00818733 | 183 | 4.01 | -7 | 0.00776311 |
| lena.gif | 840 | 3.85 | -262 | 0.00871613 | 765 | 3.85 | -166 | 0.0079379 |
| peppers.gif | 903 | 3.51 | -110 | 0.00857859 | 753 | 3.52 | -81 | 0.00715357 |

Table 2: Experimental results for several choices of the parameters

[8] WELCH, T. A. A technique for high-performance data compression. *IEEE Computer 17*, 6 (June 1984), 8–19.

[9] ZIV, J., AND LEMPEL, A. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theory 24*, 5 (Sept. 1978), 530–536.