Error-Resilient LZW Data Compression*

Yonghui Wu Stefano Lonardi
Dept. Computer Science & Engineering
University of California
Riverside, CA 92521

Wojciech Szpankowski
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Abstract

Lossless data compression systems are typically regarded as very brittle to transmission errors. This limits their applicability to domains like noisy tetherless channels or file systems that can possibly get corrupted. Here we show how a popular lossless data compression scheme used in file formats GIF, PDF, and TIFF, among others, can be made error-resilient in such a way that the compression performance is minimally affected. The new scheme is designed to be backward-compatible, that is, a file compressed with our error-resilient algorithm can be still decompressed by the original decoder. In this preliminary report, we present our scheme, collect some experimental data supporting our claims, and provide some theoretical justifications.

1 Introduction

As many practitioners of data compression know, compressed streams are very sensitive to transmission errors. Even a single error can have devastating effects and compromise all the data downstream [9]. In fact, the lack of resiliency of adaptive data compression has been a practical drawback of its use in many applications. Joint source-channel coding [6, 2] 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 the popular Lempel-Ziv-Welch (LZW) compression scheme. LZW is a lossless data compression algorithm developed by T. Welch in 1984 [11] for implementation in hardware for high-performance disk controllers as an improved version of the LZ-78 dictionary coding algorithm [12]. 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 employed in the V.42bis modem standard and can also be used optionally in TIFF images and PDF formatted documents.

Here we focus on the problem of adding error-resiliency to LZW. Compared to our previous work on LZ-77 [2], extracting from LZW/LZ-78 the extra redundancy bits needed to store the error-correcting parity bits turned out to be significantly more involved. We had two main constrains in the design of the new scheme. First, we wanted to maintain

*This project was supported in part by NSF CCR-0208709, NSF CAREER IIS-0447773, and NSF DBI-0321756.

0-7695-2545-8 /06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

the backward-compatibility with the original LZW, that is, we wanted a file compressed with the new LZW (augmented for error-resiliency) to be decodable by the original LZW. Backward-compatibility is an essential property because it allows one to deploy the new scheme gradually over the existing one, without interruptions of service. Second, we wanted the compression ratio to be minimally affected 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 small set of selected LZW phrases in the compressed stream, so that the pattern of shorter phrases would encode the extra bits. According to our experimental results, our strategy affects compression only marginally. In fact, sometimes the file compressed with the new LZW which embeds the parity bits is actually shorter than the original LZW compressed file appended with the file of the parity bits. An extensive testing of several LZW decoders available allows us to claim that our error-resilient LZW encoding is backward-compatible.

The content of this paper can be summarized as follows. Section 2 introduces some basic concepts and notation. In Section 3 we carefully detail the process of embedding extra bits in LZW, which depends on two integer parameters. In Section 4 we discuss how to choose these parameters so that we extract enough bits from the compressed stream to store the parity bits. An asymptotic analysis of the redundancy of the new scheme is carried out in Section 5. In Section 6 we describe how to achieve error-resiliency in LZW using the new LZW encoder. Finally, in Section 7 we present the implementation and discuss some preliminary experimental results.

2 Basic concepts

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

For completeness of presentation we briefly review the original LZW scheme [11, 12]. The algorithm parses the text T *online* left-to-right into *phrases*, where each phrase is the longest matching substring seen previously plus one extra symbol. Each new phrase is added to the dictionary, which is first initialized to include every single symbol in the 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 with all the symbols in the alphabet. Then, it 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.

0-7695-2545-8 /06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

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 is unique. The greediness implies lack of redundancy which in turns effectively prevents us from embedding directly extra bits into the compressed data stream. A possible solution to this problem is to relax the greediness by making some of the phrases a little shorter, in such a way to introduce enough redundancy to encode the extra bits. Relaxing the length of *too many* phrases will, however, degrade considerably the compression performance. Care must be taken to select the set of phrases to shorten so that we will have the necessary "extra space" to store the bits for error detection and correction. Note that by relaxing the greediness some entries in the dictionary will have multiple codewords associated with them.

A somewhat similar approach was taken in [10], where the authors analyze a scheme where the entries in the dictionary might have multiplicities up to a constant b. After this manuscript was completed, papers [7, 8] were brought to our attention. In these works, the parity of the length of *all* LZW phrases is matched (and possibly adjusted) in order to hide extra bits in the compressed file. According to our experimental evaluations, the latter strategy is not practical because the compression performance is degraded too much.

Before proceeding further, we first determined whether our modification in the encoder would produce a compressed file that can still be decompressed by the original LZW algorithm (called *greedy-LZW* hereafter, to distinguish it from our *relaxed-LZW* scheme). We argue that the backward-compatibility of relaxed-LZW depends on the specific data structure used to represent the dictionary in the implementation of the LZW decoder. Although in the literature the dictionary is always represented as a trie, all LZW implementations we checked use a fixed-size hash table (typically with 4096 entries). Because the decoder uses a hash table instead of a trie, it is not affected by multiple identical entries in the dictionary. The decoder refers to the existing phrases by their indices, whether there exists multiple such phrases or not, and concatenates two strings to produce the next phrase. As a consequence, our relaxed-LZW scheme is backward-compatible with the greedy-LZW decoding. We verified the backward-compatibility on various software that support LZW. For example, for GIF images, we tested MS Paint, MS Internet Explorer and Mozilla Firefox. We also tested Unix Compress and Winzip. All the latter software were able to decompress a relaxed-LZW stream. We expect PDF and TIFF decoders to be capable of handling our scheme as well.

A detailed description of the relaxed-LZW scheme is in order. Let M denote the extra bits that are going to be embedded into the compressed text T', which we call *message*. The LZW phrases that will become shorter because of the relaxation are called *non-greedy* phrases. The behavior of relaxed-LZW is determined by two non-negative integer parameters called K and L. The integer K specifies the number of bits of the message that is embedded in the interval between two consecutive non-greedy phrases, whereas the parameter L specifies the number of bits that is embedded in the length of each non-greedy phrase.

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

0-7695-2545-8 /06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

```
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) do
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.             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 1. A sketch of the encoder capable of embedding a message M while compressing text T into a LZW stream. K and L are two integer parameters (see text for details)

contents of the 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 the LZW scheme, as follows. We initialize a counter to 0, and generate new phrases greedily according to 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 increase the counter by one. As soon as the counter reaches the value k we relax the length of that phrase by l symbols. Then, the counter is reset, and a new cycle begins (see Figure 1 for details).

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, 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 according to the LZW algorithm. Note that in order to determine whether a phrase is greedy or not, the decoder need to look ahead several phrases. This can be done by saving the last few phrases in a look-ahead buffer. A sketch of the decoder is illustrated in Figure 2.

As a final remark, we want to note that the strategy described above does not exploite the multiplicities in the dictionary to embed additional bits of the message. When the relaxed-LZW algorithm looks for a longest prefix of text (to be compressed) that matches an entry in the dictionary, there might be multiple identical 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 used in [2, 3] 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.

IEEE COMPUTER SOCIETY

MESSAGE_EMBEDDING_LZW_DECODER(T', K, L)

1. $T, M, buf, i \leftarrow \emptyset, \emptyset, \emptyset, 0$
2. **while** (have not finished decoding yet) **do**
3. $\quad phrase1 \leftarrow$ decode the next phrase according to the standard LZW decoder
4. $\quad T \leftarrow T + phrase1$
5. $\quad$ Fill buf
6. $\quad phrase2 \leftarrow$ encode the text in buf according to the standard LZW encoder
7. $\quad$ **if** ($|phrase2| > 2^L$) **then**
8. $\quad\quad i \leftarrow i + 1$
9. $\quad\quad$ **if** ($|phrase1| < |phrase2|$) **then**
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 2. A sketch of the decoder capable of recovering the embedded message M from a relaxed-LZW stream T'. K and L are two integer parameters (see text for details)

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 new LZW stream tend to be slightly longer. Intuitively, the compression performance degrades as K decreases and L increases, but at the same time the number of bits embedded in the compressed text increases. In the error-resilient application it is crucial to determine the best trade-off. The first step is to compute the total number of parity bits needed, step that will be discussed later in Section 6. Once that is done, one need to estimate the number of bits that can be embedded in T' as a function of the parameters K and L. The aim is to create enough "space" for the parity bits of the error-correcting code, but not much more so that the compression will degrade.

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, if the text is long enough and L is small (in our experiments $L = 0$ or $L = 1$), the assumption will be satisfied. For simplicity, we assume that the message M to be embedded has been generated by an i.i.d. source with `0` and `1` having equal probabilities. Then, on average we will have 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. To simplify the exposition, we set $\bar{k} = (2^K + 1)/2$ and $\bar{l} = (2^L + 1)/2$.

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 consists of the greedy phrases, which are determined by T_1. According to [1], the expected number of the phrases approximately equal to $p_1 = n_1 h_1 / \log n_1$, where h_1 is the entropy of T_1. Thus, the average length of the greedy phrases will be $l_1 = \log n_1 / h_1$. The number of blocks of message that is embedded in the text will be $B = p_1/\bar{k}$, and hence we have $|M| = (K + L)B = (K + L)p_1/\bar{k}$. The average length l_2 of non-greedy phrases is $l_2 = l_1 - \bar{l}$ and therefore $n_2 = Bl_2 = \frac{p_1}{\bar{k}}(\frac{n_1}{p_1} - \bar{l}) = \frac{n_1}{\bar{k}} - \frac{n_1 h_1 \bar{l}}{\bar{k} \log n_1}$.

IEEE COMPUTER SOCIETY

From the set of equations above, we can estimate the size $|M|$ of the message given n, h_1, K, and L. The entropy h_1 can either be computed directly from the text according to its definition, or it can be inferred experimentally. In our implementation, we chose the latter method. In our estimation phase, we first count the number of phrases P generated when the text is compressed by the greedy-LZW algorithm. Then, according to the asymptotic equation $P \approx nh/\log_2 n$, we set $h = P \log_2 n/n$. Inferring h in this way has the advantage of taking into account any constant factor that may be hidden in the asymptotic equation.

One crucial step in estimating the channel capacity is to determine n_1 from the above relations. Let us define a function $f(n_1) = n_1 + \frac{n_1}{k} - \frac{n_1 h \bar{l}}{k \log n_1}$. The value n_1 we are looking for is the one that satisfies $f(n_1) = n$. Observe that f is monotonically increasing in n_1 for reasonably chosen K and L and for sufficiently large n_1, because the function is dominated by the first two terms. Therefore, a binary search can be employed to find n_1. Once n_1 is determined, we calculate $|M|$ from the equations above.

As we will show in our experimental results section, our estimation of $|M|$ is fairly accurate. The estimate is more precise when $L = 0$, which is expected since we are assuming that all phrases are longer than 2^L.

5 Asymptotic analysis of the redundancy

In this section we are concerned with the theoretical analysis of the redundancy of relaxed-LZW when the input is large ($n \to \infty$). In particular we are interested in comparing the redundancy of relaxed-LZW to that of greedy-LZW/LZ-78, to determine how much we are losing due to the embedding. We follow the notation of [4], and we assume the text to be generated by an i.i.d. source over a binary alphabet (with probabilities p and $q = 1 - p$), but the analysis can be easily generalized. Observe that on average relaxed-LZW decreases the number of manipulated symbols by $x_n \bar{l}/\bar{k}$ where x_n is the yet unknown (average) number of phrases when a string of length n is compressed.

As in [4], we define a function $\mu(x)$ that gives the length of the text as a function of the number of phrases x

$$\mu(x) = \frac{x}{h} \log x - \frac{Ax}{h} - \frac{x\bar{l}}{\bar{k}} + O\left(\frac{\log x}{h}\right)$$

where $A = 1 - \gamma - (h_2/2h) + \alpha - \delta_0(n)$ 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

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

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, when $p = q = 0.5$), and $\lim_{x\to\infty} \delta_0(x) = 0$ otherwise.

Define now x_n as

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

IEEE COMPUTER SOCIETY

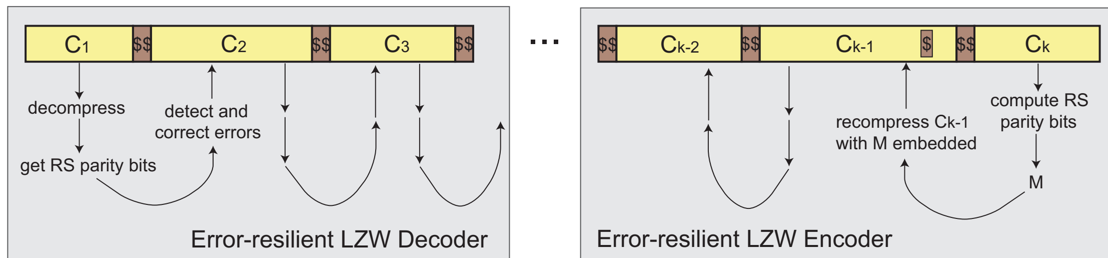

Figure 3. The sequence of operations on the chunks for the encoder (right-to-left) and the decoder (left-to-right)

that is, n is the length of the text when number of phrases is x_n. Let us define M_n to be the random variable associate with number of phrases constructed by relaxed-LZW when given a text of length n. Observe that the average number for phrases of our scheme is $E[M_n] \sim x_n$ and therefore the code length L_n of our scheme is

$$L_n = x_n(1 - \bar{l}/\bar{k})[\log(x_n(1 - \bar{l}/\bar{k})) + 1].$$

Thus, the relative average redundancy r_n becomes

$$\begin{aligned} r_n &= \frac{x_n(1 - \bar{l}/\bar{k})[\log(x_n(1 - \bar{l}/\bar{k})) + 1] - nh}{n} \\ &= h\frac{1 + A + \bar{l}h/\bar{k}}{\log n}. \end{aligned}$$

Comparing it to the regular LZW/LZ-78, the redundancy of the relaxed-LZW is increased by $\bar{l}h/(\bar{k}\log n)$.

6 Error-resilient LZW

The last piece of the puzzle is to show how to use the extra space to embed the parity bits to achieve error-resiliency. As in [2], we can use Reed-Solomon codes [5] to detect and correct errors. Reed-Solomon codes are block-based error-correcting codes widely used in digital communications and storage. Compared to our error-resilient LZ-77, turning the relaxed-LZW into a error-resilient scheme is somewhat more involved.

The operations performed by the error-resilient LZW encoder and the decoder are illustrated in Figure 3. First, the text T is compressed with the greedy-LZW. As mentioned above, the typical LZW implementation has a fixed sized dictionary (implemented as an hash table). As soon as the dictionary contains 4096 entries, the dictionary is flushed out and refreshed. In the figure, we denote the places in T' where the dictionary is refreshed by the symbol $\$$. In LZW files these position are marked by a special end-of-dictionary (EOD) symbols. The compressed text T' can therefore be broken into *chunks* $C_1, C_2, \ldots, C_k$.

The error-resilient LZW encoder processes the chunks right-to-left, starting from the last chunk. In the generic step, it computes the Reed-Solomon parity bits on the chunk C_i and then embeds the parity bits in the chunk C_{i-1}. The embedding requires the encoder to re-compress the chunk C_{i-1} by calling MESSAGE_EMBEDDING_LZW_ENCODER(). Because we are relaxing the length of some of the phrases, that may introduce in C_{i-1} a

0-7695-2545-8 /06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

	$\lvert T'\rvert$	l	$\lvert T'_M\rvert$	$\lvert M\rvert$	l_M	$\lvert T'_M\rvert - \lvert T\rvert - \lvert M\rvert$	$\lvert M\rvert/\lvert T'\rvert$	estim. $\lvert M\rvert$
`airplane`	64908	5.77	66468	1706	5.63	-146	0.02628	1980
`baboon`	149414	2.49	151804	2169	2.45	221	0.01451	4678
`couple`	19604	4.88	20088	505	4.77	-21	0.02576	587
`girl`	23573	4.04	24127	566	3.94	-12	0.02401	712
`lena`	96373	3.87	98770	2396	3.78	1	0.02486	2973
`peppers`	105262	3.54	107792	2372	3.46	158	0.02253	3258

Table 1. Comparing the size of a set of 8 bpp GIF images compressed with the greedy-LZW against the relaxed-LZW scheme ($K = 5, L = 1$). All pictures are 512×512 except `couple` and `girl`, which are 256×256 pixels. $|M|$ is the length of the message in bytes that is embedded in the GIF file. $|T'|$ is the length of the original GIF file in bytes, $|T'_M|$ is the length of the GIF with the message embedded. l is the average length of the LZW phrases before the embedding and l_M is average phrase length after the embedding. The last column reports our estimate of the message length.

premature end of dictionary. For this reason, we must use two back-to-back EOD symbols to encode the boundaries of the chunks. This will not affect the backward-compatibility.

The error-resilient LZW decoder processes the chunks left-to-right, starting from C_1. In the generic step, it decompresses chunk C_i with the MESSAGE_EMBEDDING_LZW_DECODER(), which also retrieves the Reed-Solomon parity bits. The parity bits allow the decoder to detect and correct errors in C_{i+1} before that chunk is decompressed. Note that the first chunk is not protected against errors, but not much can be done unless we sacrifice the backward-compatibility.

7 Experimental Results

Although a full implementation of the error-resilient LZW is still in progress, we have implemented both MESSAGE_EMBEDDING_LZW_ENCODER and MESSAGE_EMBEDDING_LZW_DECODER in C++, and tested them on GIF files (which are internally compressed by LZW). Our implementation consists of two executables, called `relaxedlzw` and `messagedecoder`. The executable `relaxedlzw` takes as input a standard GIF file, a message file and the parameters K and L, and produces a message-embedded GIF file. The encoder also provides some useful statistics, such as the total number of phrases, the length of the message that has been embedded, etc. The executable `messagedecoder` takes the message-embedded GIF file and the parameters K and L (which should match the ones used in the encoding) as input. It regenerates the message-free GIF file and extracts the message that was embedded.

To establish the backward-compatibility, we have extensively verified that the message-embedded GIF files can be opened with several existing software capable of displaying GIF images (e.g., MS Paint, MS Internet Explorer and Mozilla Firefox). We have also verified that the GIF file extracted by `messagedecoder` is identical to the original GIF file, and that the message extracted from the relaxed-LZW file is identical to the one that was embedded. The source code of the encoder and the decoder is available in the public domain at `http://www.cs.ucr.edu/~yonghui`. We have also implemented separately in Python the estimation of the size of the message given the parameters K, L and the file to be compressed.

IEEE COMPUTER SOCIETY

	$\lvert M\rvert$	l_M	$\lvert T'_M\rvert - \lvert T\rvert - \lvert M\rvert$	estim. $\lvert M\rvert$	$\lvert M\rvert/\lvert T'\rvert$	$\lvert M\rvert$	l_M	$\lvert T'_M\rvert - \lvert T\rvert - \lvert M\rvert$	estim. $\lvert M\rvert$	$\lvert M\rvert/\lvert T'\rvert$
	$K = 1, L = 0$					$K = 1, L = 1$				
airplane	5295	3.91	25058	2532	0.081577	9122	3.97	20001	5249	0.14053737
baboon	10973	1.66	62620	6470	0.07344023	10768	1.97	28233	14201	0.07206821
couple	1509	3.38	7035	762	0.07697408	2650	3.39	5798	1591	0.13517649
girl	1823	2.76	8798	937	0.07733423	2978	2.85	6654	1975	0.12633097
lena	8003	2.53	42706	3912	0.08304193	12624	2.65	31451	8271	0.13099104
peppers	8392	2.34	45190	4320	0.07972487	12278	2.51	30728	9188	0.11664228
	$K = 2, L = 0$					$K = 2, L = 1$				
airplane	5227	4.73	8759	3493	0.08052936	6799	4.75	6950	5370	0.10474825
baboon	10778	2.02	23867	8654	0.07213514	8365	2.2	11363	13807	0.05598538
couple	1524	4	2638	1045	0.07773923	1979	4.01	2134	1614	0.10094878
girl	1794	3.34	3023	1278	0.07610401	2207	3.4	2202	1987	0.09362406
lena	7776	3.12	15101	5337	0.08068649	9337	3.18	11384	8314	0.09688398
peppers	8167	2.87	16189	5875	0.07758735	9119	2.97	11055	9188	0.08663145
	$K = 3, L = 0$					$K = 3, L = 1$				
airplane	3947	5.23	2720	3239	0.06080914	4542	5.24	2035	4384	0.06997596
baboon	8100	2.23	8947	7821	0.05421178	5718	2.33	4477	10821	0.0382695
couple	1160	4.45	779	964	0.05917159	1289	4.48	521	1309	0.06575188
girl	1356	3.67	995	1174	0.05752343	1468	3.69	690	1600	0.06227463
lena	5813	3.47	5311	4903	0.06031772	6205	3.49	4031	6690	0.06438525
peppers	6153	3.17	5934	5383	0.05845414	6185	3.23	3967	7362	0.05875814
	$K = 4, L = 0$					$K = 4, L = 1$				
airplane	2656	5.48	623	2450	0.04091945	2819	5.49	366	3090	0.0434307
baboon	5432	2.36	3033	5814	0.03635536	3698	2.4	1482	7418	0.02475002
couple	778	4.67	140	727	0.03968577	828	4.67	93	919	0.04223627
girl	940	3.83	282	882	0.03987612	952	3.84	201	1117	0.04038518
lena	3873	3.66	1555	3686	0.0401876	3941	3.67	1129	4667	0.04089319
peppers	4071	3.35	1718	4039	0.03867492	3917	3.37	1176	5121	0.0372119

Table 2. Experimental results for $1 \leq K \leq 4$ and $0 \leq L \leq 1$ (see caption of Table 1 for the legend)

For our tests, we used images which are commonly used in digital image processing and data compression literature. Table 1 shows our experimental results for $K = 5$ and $L = 1$. The column $|M|$ indicates how many extra bytes (not bits) we can embed into the original GIF file. Since we are still missing the module that computes the Reed-Solomon parity bits, the extra bits were randomly generated by an i.i.d. model with equal probability of 0 and 1. Column $|T'_M| - |T| - |M|$ determines whether the relaxed-LZW packs more information than the greedy-LZW. With respect to the latter measure, our scheme performs better on `airplane` and `couple` than the rest of the GIF files, and `baboon` is the worst among all of them. This is because the average phrase length for `airplane` and `couple` (5.77 and 4.88 respectively) is much higher than that for `baboon` (2.49), and thus reducing the length of the phrases for the first two images will not have such a negative impact on the compression than for `baboon`. Column $|M|/|T'|$ gives the number of bytes embedded for each byte in the compressed file. For example, on the file `airplane` we can embed one byte every 38 bytes of compressed text, on average. We notice that the variation of $|M|/|T'|$ among the files is quite small, except `baboon`, which we believe is due to its relatively poor compression ratio. We also report our estimation of $|M|$, which is quite accurate, in particular when $K > 4$ (again, with the exception of `baboon`).

Tables 2 and 3 summarize our experimental results for several choices of parameter K and L. As expected, the number of extra bits that can be embedded (i.e., $|M|$) decreases as the parameter K increases. However, the relationship between $|M|$ and the parameter L is not as clear. In our observations, when the parameter K is small, say less than 4, increasing the parameter L from 0 to 1 usually increases the number of bits that can be embedded as well. However, increasing L beyond 1 neither helps to embed more bits nor improves the image compression ratio.

0-7695-2545-8 /06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

	$\|M\|$	l_M	$\|T'_M\|-\|T\|-\|M\|$	estim. $\|M\|$	$\|M\|/\|T'\|$	$\|M\|$	l_M	$\|T'_M\|-\|T\|-\|M\|$	estim. $\|M\|$	$\|M\|/\|T'\|$
	$K=5, L=0$					$K=5, L=1$				
airplane	1663	5.63	-90	1643	0.02562087	1706	5.63	-146	1980	0.02628335
baboon	3413	2.42	756	3856	0.02284257	2169	2.45	221	4678	0.01451671
couple	479	4.77	-27	486	0.02443378	505	4.77	-21	587	0.02576004
girl	582	3.92	63	589	0.02468926	566	3.94	-12	712	0.02401052
lena	2441	3.76	276	2461	0.02532867	2396	3.78	1	2974	0.02486173
peppers	2566	3.45	357	2694	0.02437726	2372	3.46	158	3258	0.02253424
	$K=6, L=0$					$K=6, L=1$				
airplane	987	5.72	-300	1023	0.01520613	982	5.71	-181	1196	0.0151291
baboon	2039	2.46	-130	2387	0.01364664	1305	2.47	-98	2800	0.00873412
couple	288	4.81	-14	302	0.01469087	286	4.83	-64	354	0.01458885
girl	329	3.99	-37	366	0.01395664	311	3.98	-4	428	0.01319305
lena	1478	3.82	-236	1529	0.01533624	1390	3.83	-260	1790	0.01442312
peppers	1545	3.49	-84	1673	0.01467766	1376	3.49	25	1959	0.01307214
	$K=7, L=0$					$K=7, L=1$				
airplane	591	5.75	-303	608	0.00910519	583	5.75	-294	696	0.00898194
baboon	1184	2.47	-127	1415	0.00792429	782	2.48	-102	1621	0.00523377
couple	182	4.84	-38	180	0.00928381	175	4.85	-37	206	0.00892674
girl	193	4.01	-22	217	0.00818733	183	4.01	-7	249	0.00776311
lena	840	3.85	-262	908	0.00871613	765	3.85	-166	1039	0.0079379
peppers	903	3.51	-110	993	0.00857859	753	3.52	-81	1137	0.00715357
	$K=8, L=0$					$K=8, L=1$				
airplane	359	5.76	-172	350	0.0055309	309	5.76	-132	395	0.00476058
baboon	706	2.48	-65	815	0.00472512	407	2.48	-100	918	0.00272397
couple	105	4.86	-45	103	0.00535604	100	4.87	-47	116	0.00510099
girl	122	4.01	10	125	0.00517541	111	4.02	-4	141	0.00470877
lena	498	3.86	-260	523	0.00516742	436	3.86	-245	589	0.00452408
peppers	517	3.52	-99	572	0.00491155	450	3.52	-25	644	0.00427504

Table 3. Experimental results for $5 \leq K \leq 8$ and $0 \leq L \leq 1$ (see caption of Table 1 for the legend)

References

[1] JACQUET, P., AND SZPANKOWSKI, W. Asymptotic behavior of the Lempel-Ziv parsing scheme and digital search trees. *Theor. Comput. Sci. 144*, 1–2 (June 1995), 161–197.

[2] 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.

[3] 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.

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

[5] REED, I. S., AND SOLOMON, G. Polynomial codes over certain finite fields. *J. SIAM 8* (1960), 300–304.

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

[7] SHIM, H. J., AHN, J., AND JEON, B. DH-LZW: lossless data hiding in LZW compression. In *International Conference on Image Processing, 2004. ICIP '04* (2004), vol. 4, pp. 2195– 2198.

[8] SHIM, H. J., AND JEON, B. DH-LZW: Lossless data hiding method in LZW compression. In *Advances in Multimedia Information Processing (PCM 2004), Lecture Notes in Computer Science* (2004), vol. 3333, pp. 739–746.

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

[10] 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.

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

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

0-7695-2545-8 /06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY