A lossy data compressor based on the LZW algorithm.*

V.G. Ruiz and I. García
Dept. Arquitectura de Computadores y Electrónica.
University of Almería, 04120-Almería, Spain.

Abstract

LZW (Lempel-Ziv Welch) is a lossless data compression algorithm which eliminates the intercharacter redundancy by searching linear patterns in an input data stream. Our proposal is a variation of the original LZW encoder which performs a fast lossy data compression by the use of a quantized hashing search technique. For two-dimensional data streams (images) a Recursive-Z Ordering is proposed.

1 Introduction

Data compression is the process of eliminating or reducing the redundancy in the data representation in order to achieve savings in storage and communication costs. Data compression techniques can be classified into two categories: lossless and lossy schemes. Using **lossless** methods, the exact original information is obtained while for **lossy** schemes a close approximation of the original information can only be recovered. Lossless algorithms can be used to encode any source of information, from a text data file to a video sequence, while lossy algorithms can only be used to compress raster data, such as sampled sound, raw images or video. The choice of a data compressor category (lossless or lossy) depends on the application context. For instance, lossless methods are appropriate for specialized applications such as medical imaging or satellite photography where reliability of the image reproduction is a critical factor. On the other hand, lossy compression methods are useful in applications such as telephone transmission or digital television, for which better compression ratios are obtained but at the price of paying a loss of i ormation.

LZW (Lempel-Ziv Welch) is a lossless data compression algorithm which eliminates the intercharacter redundancy by searching linear patterns in the input data stream. Typical applications based on LZW are the standard `compress` utility of UNIX and the GIF image format [1]. LZW works by replacing an occurrence of a group of bits in a piece of data with reference to a previous occurrence in the set of no compressed data [6]. The LZW encoder [4] makes use of a string table to search for an input string of characters w. The source of the uncoded characters is named the **data stream**. In each iteration, the encoder concatenates the w string and the next input character k, building

*This work was supported by the Ministry of Education of Spain (CICYT TIC96-1125-C03-03 and CICYT TIC96-1259-E)

pression code w to the **code stream** (encoder's output) and makes $w \leftarrow k$. Otherwise, the encoder makes $w \leftarrow$ address of (w, k) and tries with the next input character. The beauty of LZW scheme consists in the decoder only needs the compressed code stream to rebuild the string table and the original data stream.

The harder work for the encoder is the search for a string (w, k) in the string table. This task can be performed by a binary search [5] or through hashing. The binary search is the optimal solution for obtaining the best compression ratios (see equation 1) while Hashing is faster but obtaining worst compression ratios depending of the collision ratio. However speed up and CR for the hashing search will depend on the choice of the hash function.

Our proposal is a variation of the original LZW encoder. In order to reach a better compression ratio it performs a fast lossy data compression using a approximate but fast hashing search technique. These approximate searches consist in a quantization of the searching key in the string table. It will be called *Quantized Hashing Search* (QH) and described in section 2. On the other hand, the decoder is the typical LZW decoder, with no variations.

Intended to eliminate redundancies due to the correlation between neighbor data for two-dimensional data (images), a new ordering of the input data stream is proposed in our implementation. It is called *recursive-Z Ordering* and its main advantage consists in the reduction of some kind of artifact appearing in the standard *Row Ordering*. It works in a similar way to the well known 8×8 *Zig-Zag Ordering* [2]. The recursive-Z Ordering is described in section 2.

LZW algorithm are shown. Functions used in the evaluation of our proposal are the *C*ompression *R*atio (CR) and the *S*ignal-to-*N*oise *R*atio (SNR) defined as:

$$CR = \frac{Size(x) - Size(z)}{Size(x)} \times 100 \quad (1)$$

$$SNR = 10 \times \log_{10} \frac{\sum_{i=1}^{Size(x)} x_i^2}{\sum_{i=1}^{Size(x)} (x_i - y_i)^2} \quad (2)$$

where x and z are the original and compressed data streams, respectively, and y is the restored (compressed and uncompressed) data stream.

2 Software implementation

The LZW encoder is a greedy algorithm which in every iteration process a input character, so, the complexity of the encoder is linearly dependent of the size of the input data stream.

The main task during an iterative step is the search for a pair (w, k) in the string table. Obviously, the choice of the searching algorithm rebounds directly on the execution time of the encoder. The faster alternatives to performs the search are:

Binary search is the most suitable when the size of the string table is small. The binary search shows a complexity of $O(\log_2(N))$, where N is the number of items stored in the search tree (see figure 2). The main advantage of this scheme is the optimal exploitation of the reserved memory for the string table. This is because of all the codes in the string table are stored in a block of consecutive cells.

Hashing is the fastest scheme to find a pair (w, k) in the string table. Theoretically, most of the times, a pair can be

sions are allowed. A suboptimal exploitation of the string table is made; i.e. the string table is almost a sparse table. This problem causes longer code streams than that obtained by the binary search. Therefore, an important aspect is the selection of the particular hash function which transforms the item to be searched in an entry of the hash table. This function should spread the keys along the hash table uniformly and must be fast [6]. In case of the hash function doesn't find the correct pair, it is said that a collision problem has appeared. To overcome a collision problem two options can be tried: (a) to look for an alternative entry in the string table using linear probing, clustering, increment functions, etc [3], but spending extra time. (b) the dumb alternative: if a collision exists, it can be supposed that the pair (w, k) is not in the string table and an output code will be sent. For this reason, in order to minimize the size of the output code it is very important to minimize the number of collisions by the selection on an adequate hash function. The hashing function for our implementation is:

$$hash \leftarrow w \oplus (k \ll (\mathtt{W} - 8)) \qquad (3)$$

where $\mathtt{W} = \log_2(\text{string table size})$.

Quantized hashing is the new characteristic of the our lossy LZW algorithm and consists in searching for a pair (w, k) using the most significant bits of the pair only. Usually, in 8-bit compressions w is a 16 bits unsigned integer and k is a 8 bit character. The problem of finding the pair (w, k) is similar to that of searching for a number $w * 256 + k$. It can be seen that w codifies the string of characters which has been recognized previously. Obviously, we must find the entry in the string table which contains the suitable w component, but it is not so important to obtain a exact search in the string table because it only represents the last input character of the character stream. To perform an approximate search of the binary number $w * 256 + k$, we can despise the lower significant bits of k. It produces a minimal distortion for the search. The equation used to perform the approximate search is the next quantized hash function:

$$hash \leftarrow (w \ll \mathtt{P}) | (k \gg (8 - \mathtt{P})) \qquad (4)$$

where $\mathtt{P}$ is the number of bits retained in k. The number of output codes is small and higher compression ratios can be reached.

In addition to the searching method, it is important to analyze the criteria used in the selection of the size of the string table and the usefulness of using a variable output code size. It must be taken into account that the memory of any computer is finite as well as that the efficiency of the memory hierarchy is better for smaller data structures. Both aspects will rebound in the encoder execution time. For example, for a hashing search with no collisions, any entry to the string table can be found by one single access. Depending on its size the string table could be kept on the cache, main or secondary memory. Therefore the smaller the string table the shorter the execution time will be.

Related to the output code size, it must be taken into account that computers easily manage bytes or words but it spends extra time in computations on codes whose lengths are not proportional to 8 bits. However in order to reach a better compression ratio the output code at any time must be as small as possible and therefore the use of a variable output code is an important matter. It seems to be worth the effort of using a variable output code size and so to get a better CR.

Figure 1: The Z-order of a squared image with 4×4 pixels

Finally, in our implementation of the LZW, an ordering of the input data stream called Recursive-Z Ordering has been implemented as well as the well known 8×8 Zig-Zag Ordering (used in the JPEG compression standard) and the natural Row Ordering. The use of these reorderings will be useful only to compress two-dimensional array of data. The usefulness of a Zig-Zag Ordering or Recursive-Z Ordering is not only due to the compression ratio is slightly improved but also because they avoid some kind of artifacts appearing in lossy compressors that make use of a Row Ordering of the input data stream. In table 1 some results related to this topics are shown.

The Recursive-Z Ordering of a image divides the image in four squares of equal size and run the blocks following a `Z` shape. It is applied recursively, until every block only contains one pixel. An example of this recursive procedure is shown in the figure 2.

3 Evaluation

This section shows the compression measurements tested for an image (lena) using several values for the string table size, the three ordering schemes described above and several levels of our approximated hash search. Figure 2 shows the computation time for the three implementations of the LZW encoder (using binary search, hashing, and quantized hashing) as a function of the string table size (the range of the string table size is relatively small). It can be seen that QH-LZW is the faster solution for large string tables.

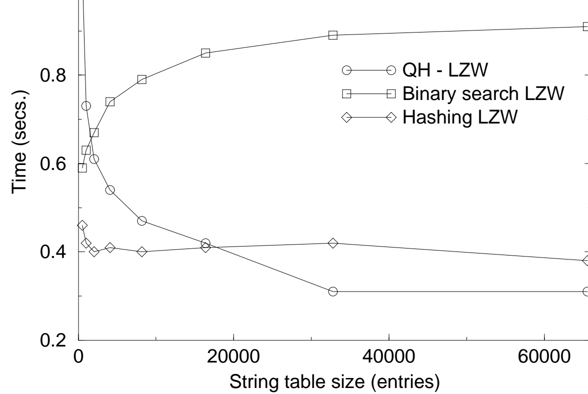

Figure 2: The search complexity (LZW)

As an example of our results, in table 1 some values of CR, SNR and time are shown. It is clear that the compression ratio (CR) using QH-LZW with `P`=7 is much better than that obtained for UNIX `compress` and at the same time, it keeps a very high level for the SNR. Also, it can be seen that the smaller values of `P` the better CR and the faster compression are got. Nevertheless values for SNR decrease as the CR parameter increases. Also in table 1 it can be seen the slightly improvements introduced by the Recursive-Z (R-Z) Ordering compared to the 8×8 Zig-Zag (Z-Z) and Row Ordering (R-O), however the SNR and time are similar.

Finally figure 3 is an example of the compressed and uncompressed image of lena using the quantized hashing LZW with `P`=3.

Figure 3: Compressed 'lena' at 3 bits/pixel, $SNR = 53.5dB$ (P=3).

References

[1] (c) Compuserve Incorporated. *Graphics Interchange Format (GIF) Specification*, June 15, 1987.

[2] Gregory K. Wallace. The JPEG Still Picture Compression Standard. *IEEE Transactions on Consumer Electronics*, December 1991.

[3] Robert L. Kruse - Bruce P. Leung - Clovis L. Tondo. *Data Structures and Program Design in C.* Prentice-Hall, 1991. ISBN: 0-13-726332-5.

[4] Terry A. Welch. A Technique for High-Performance Data Compression. *IEEE Computer*, pages 8–19, June 1984.

[5] V. G. Ruiz, I. García. Una Implementación del algoritmo de compresión Lempel-Ziv Welch. In J. M. Troya Linero y C. Rodriguez Leon, editor, *Actas de I Jornadas de Informática*, 1995.

[6] Williams, R.N. An extremely fast Ziv-Lempel data compression algorithm. *...ence*, pages 362–71, Snowbird, UT, USA, 8-11 April 1991. IEEE Comput. Soc. Press, Los Alamitos, CA, USA.

Table 1: Comparison of some algorithms based on LZW and our approach for several values of the quantization parameter. Results have been obtained from Lena (256×256) using a small size string table (4096×16 bit entries). (QH = Quantized Hashing, `P` = number of bits kept in the searching key)

ALGORITHM	**RO**	**Z-Z**	**R-Z**
	CR		
UNIX `compress`	1.5	2.2	0.5
QH-LZW (P=7)	5.7	5.5	7.7
QH-LZW (P=6)	19.3	19.3	21.1
QH-LZW (P=5)	33.3	33.6	34.2
QH-LZW (P=4)	46.9	47.4	48.6
QH-LZW (P=3)	58.9	59.7	62.2
	SNR		
UNIX `compress`	∞	∞	∞
QH-LZW (P=7)	112.6	112.5	112.2
QH-LZW (P=6)	94.4	94.2	94.2
QH-LZW (P=5)	79.2	78.9	79.0
QH-LZW (P=4)	65.6	65.3	65.1
QH-LZW (P=3)	52.6	53.1	53.5
	time		
UNIX `compress`	0.48	0.48	0.45
QH-LZW (P=7)	3.66	3.65	3.62
QH-LZW (P=6)	1.73	1.70	1.69
QH-LZW (P=5)	0.96	0.89	0.87
QH-LZW (P=4)	0.55	0.54	0.56
QH-LZW (P=3)	0.41	0.38	0.40