THE RELATIVE EFFICIENCY OF DATA COMPRESSION BY LZW AND LZSS

Yair Wiseman[1*]

*[1] *Computer Science Department, Bar-Ilan University, Ramat-Gan 52900, Israel*
Email: wiseman@cs.huji.ac.il, http://www.cs.biu.ac.il/~wiseman

ABSTRACT

We explore the use of the LZW and LZSS data compression methods. These methods or some versions of them are very common in use of compressing different types of data. Even though on average LZSS gives better compression results, we determine the case in which the LZW perform best and when the compression efficiency gap between the LZW algorithm and its LZSS counterpart is the largest.

Keywords: Data Compression, Compression efficiency, LZW, LZSS

1. INTRODUCTION

Compression techniques are often partitioned into statistical and dictionary techniques (Bell, Cleary, & Witten, 1990). Huffman codes (Huffman, 1952) or Arithmetic codes (Howard & Vitter, 1992) are usually given as an example for statistical techniques, while Lempel-Ziv methods are usually given as an example for dictionary techniques. Usually Lempel-Ziv methods are adaptive (Bell, Witten, & Cleary, 1989); however, a static version of Lempel-Ziv can also be found (Schuegraf & Heaps, 1974).

This paper will concentrate on the adaptive Lempel-Ziv methods, which are also partitioned into two fundamental groups. The first group is based on reference to recurring data in the compressed file. This approach was first introduced as LZ77 (Ziv & Lempel, 1977), and versions of this approach are still commonly used by many commercial software products such as gzip (Free Software Foundation, 1991). The second group creates a dictionary of common phrases in the data. This approach was first introduced as LZ78 (Ziv &Lempel, 1978). Versions of this approach are also very common in use as a separate method such as *"compress"* of UNIX and as a part of more complicated techniques such as GIF (Willard, Lempel, Ziv, & Cohn, 1984).

One of the most popular versions of LZ77 is LZSS (Storer & Szymanski, 1982), while one of the most popular versions of LZ78 is LZW (Welch, 1984). The aim of this paper is to compare the compression efficiency of LZSS and LZW.

2. THE RELATIVE EFFICIENCY OF LZW AND LZSS

In LZSS, the encoded file consists of a sequence of items, each of which is either a single character or a pointer of the form (*offset, length*), which replaces a string of *length* characters that appeared as *offset* characters earlier in the file. Decoding of such a file is thus a very simple procedure. For the encoding, however, there is a need to locate the longest recurring strings, for which sophisticated data structures such as hash tables or binary trees have been suggested.

In LZW, the encoded file consists of a sequence of pointers to a dictionary. Each pointer replaces a string of the input file that appeared earlier and has been put into the dictionary. Encoder and decoder must therefore construct identical copies of the dictionary. In LZW, the dictionary is dynamically

constructed in a manner that can be precisely redone by the decoder. LZW finds the longest string in the dictionary. If the string $x_1,...,x_n$ is in the dictionary, but the string $x_1,...,x_{n+1}$ is not in the dictionary, LZW will add the string $x_1,...,x_{n+1}$ to the dictionary and will use the string $x_1,...,x_n$ for the current pointing.

In point of fact, both of the methods are based on *pointers*. In LZSS the pointers are the *offset* component, while in LZW, the pointers are the dictionary's pointers. Figure 1 shows how the modification of the pointer size will affect the compression efficiency in both of the methods. The test has been conducted using the King James Bible.

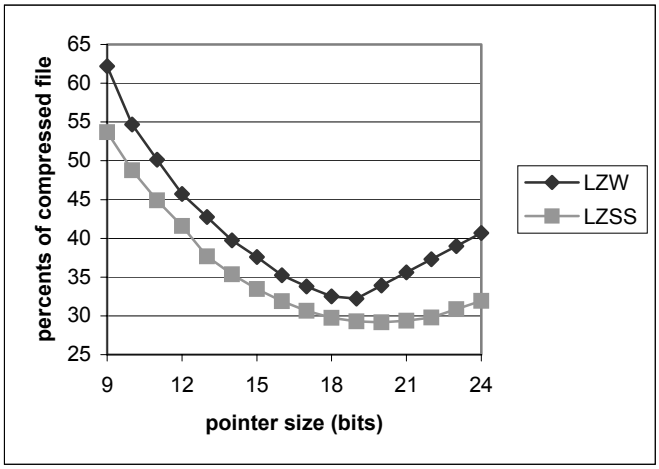

Figure 1

When the pointer size is too small, the possibilities to point are very limited, while a bigger pointer size takes too much space and can harm the compression efficiency.

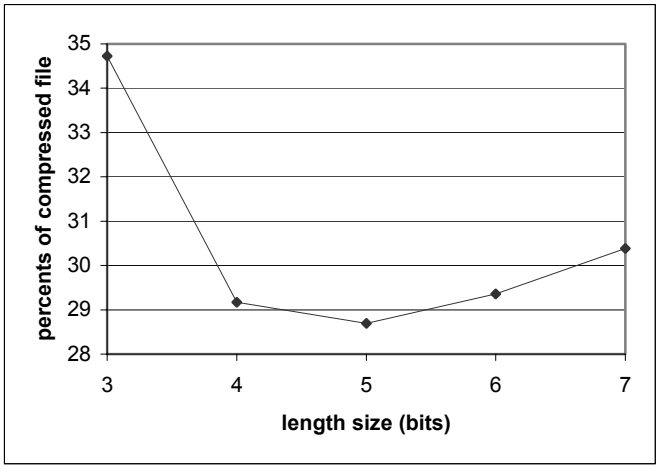

Figure 2

In LZSS there is also another parameter - the size of the *length* component. Figure 2 shows the effect of the *length* component on the LZSS compression efficiency. When the size of the *length* component is too small, only a few characters can be pointed, while a longer *length* component can be a waste because the recurring strings are usually not so long.

In Figure 3 both of the compression methods have been configured to their best parameters according to Figures 1 and 2, i.e. LZSS with 20 bits pointer plus 5 bits for the *length* component and LZW with 19 bits for the pointer. The tested files are from the Calgary Corpus (Bell, Witten, & Cleary, 1988).

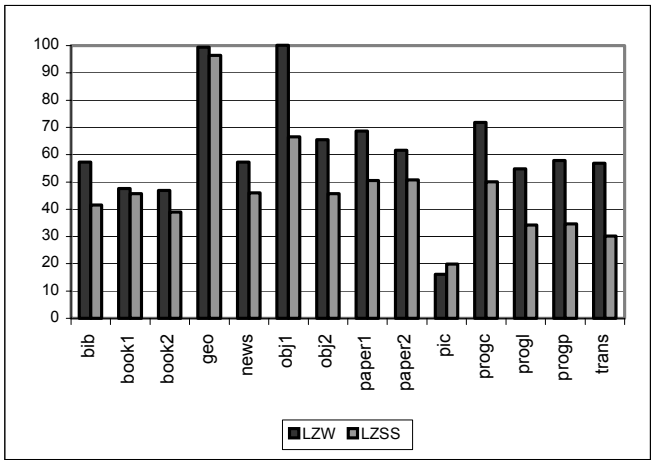

Figure 3

It can be clearly seen in both Figure 1 and Figure 3 that LZSS usually gives better results. However, when compressing the *pic* file, LZW gives better results. The reason for this exception is the uncommon content of this file, which contains a lot of nulls. When the *length* component of LZSS is just 5 bits, LZSS can put a pointer to no more than 32 bits. LZW, however, can theoretically have a pointer to an infinite string. When using LZW, each entry has an old string and a new letter; hence it can save much longer strings.

3. HOW LZW CAN OUTPERFORM LZSS

We would like to determine how good LZW can be. As shown above, LZW compresses the *pic* file better; however, LZSS can easily do better. If we significantly enlarge the *length* component, so a longer string can be pointed, LZSS can compress much better than LZW, e.g. X(1,1000) can be recursively interpreted as a long string of 1001 Xes. Therefore, our aim is to find a case that the adjustment of the *length* component size cannot help out.

Let us look at LZW with a pointer of 9 bits and build a file that contains the 256 possible characters of ASCII. This will put the pairs (0,1) (1,2) .. (254,255) into the dictionary of LZW, according to the algorithm of LZW (Horspool, 1991) (Tao & Mukherjee, 2004). If we put a 0 after the last 255, the pair (255,0) will be added, and these numbers will use up the entire dictionary. The dictionary has 512 entries. The first 256 entries are of the single characters, while the other 256 entries are of the pairs. LZW can handle these pairs better because LZSS saves one bit for character/pointer flag, several bits for the pointer, and several bits for *length* component. Usually LZSS does not bother to replace a pair of characters by a pointer because of the high price, but even if LZSS replaces the pair, the gain will be small, while LZW can save more bits when pointing just to a pair of characters.

In order to obtain the maximal difference between LZW and LZSS, we put these pairs in the file in different orders. However, we are not able to put only two pairs in adjacent locations in two strings. The second appearance of the adjacent pairs will make LZSS see the two pairs as a quadruplet, hence using just one pointer for the quadruplet. In such a case we say that the second pair is an immediate subsequent. Therefore, we must not allow it to appear again after its first occurrence.

Claim 1

Let m and n be integers. Let $\{x_1,...,x_n\}$ be a set of sequences of size m. There are just three different ways to arrange $\{x_1,...,x_n\}$ if any x_i is not allowed to be an immediate subsequent sequence to x_j in three of the different strings of sequences for every pair of $1 \leq i \leq n$ and $1 \leq j \leq n$.

Proof

By induction on the number of sequences in the set.

For n=4, there are 24 possibilities to arrange $\{x_1,x_2,x_3,x_4\}$ However, when checking all of the possibilities, it can be easily found that there are only three strings of sequences that can fulfill the stipulation that x_i is not allowed to be an immediate subsequent sequence to x_j for every pair of $1 \leq i \leq n$ and $1 \leq j \leq n$. There are some pairs of strings that can be chosen, but the number of the strings will always be not more than three, e.g. $\{x_1,x_2,x_3,x_4\}$, $\{x_3,x_1,x_4,x_2\}$ and $\{x_3,x_2,x_4,x_1\}$, so the proof is by checking all of these cases one by one.

Remark: The claim is also true for n=1,n=2,n=3. However, in order to make the proof simpler in the next stage, we used n=4.

Assume the claim is true for n-1, i.e. the set $\{x_1,...,x_{n-1}\}$ has just three ways to be arranged according to the above stipulation. Let us add the sequence x_n. The sequence x_n can be put in each of the three possible strings just once:

- At the end or at the beginning.
- After one sequence from the end or from the beginning.
- After two or more sequences from the end or from the beginning.

In all of the above cases, at least two sequences at the end or at the beginning will remain adjacent; hence there is just one possibility of adding the new sequence, and the outcome is that $\{x_1,...,x_n\}$ can be arranged in only three ways. Q.E.D.

Accordingly, a file can be built containing the single characters and three strings of pairs as described above. Such a file will be compressed by LZW with a 9 bits pointer, always in a better manner than by LZSS. The exact ratio is dependent on the pointer size of LZSS. Indeed, a 9 bits pointer size is uncommon. Therefore, a more attractive question is how the file can be enlarged to fit a larger LZW dictionary with an n bits pointer.

If LZW reads pairs, it will add triplets to the dictionary. Because there are three sequences of pairs with each sequence being 256 pairs, LZW will add 768 entries of triplets to the dictionary. As a result, the dictionary will have 1280 entries, which fits an 11 bits dictionary.

Claim 1 asserts there are only three strings of triplets that can be added, i.e. two triplets cannot be adjacent in the same order in more than one string. Moreover, because the claim is true for all the sequences of any size, the file can be enlarged by triplets, quadruplets etc., such that it will have $256 \cdot 3^{m-2}$ items of each sequence in length of m. let us name such a file SPT (stands for Single, Pair, Triplet, etc.). In these cases, even when LZSS uses a pointer, LZW will also be able to use a pointer too. Because LZW pointers are cheaper, the compression efficiency of LZW will be better.

Claim 2

The ratio between the compressed files constructed by LZW and LZSS is the highest in SPT files .i.e., there is no case where LZW can gain a better margin over LZSS.

Proof

Let us look at the number of pointers in both LZSS and LZW. In a file generated in SPT form, the number of the pointers will be equal when the file is compressed by either LZSS or LZW. Let us assume there is a better case. In this better case the number of the pointers of LZW can be less than, greater than, or equal to the number of the pointers of LZSS. Let us analyze each of these cases:

LZW creates fewer pointers: LZW only has pointers by definition, and any string pointed to by LZW can be pointed to by LZSS as well. Therefore, there is no benefit.

LZW creates the same number of pointers as LZSS: This case will yield the same ratio as the file in the above form, so no benefit is gained.
LZW creates more pointers: This case will yield a poorer result for LZW because LZW always puts a pointer, while LZSS uses pointers only in the appropriate cases. If LZSS creates fewer pointers, it will indicate that LZSS has chosen not to put a pointer because it is less expensive. In contrast, LZW puts a pointer because this is its usual behavior, and that pointer is more expensive.

Therefore, LZW outperforms LZSS best in this case. Q.E.D.

4. CONCLUSION

Usually LZSS can compress data in a better manner than LZW. If we have a long sequence of the same character, LZSS can compress it in a constant few bytes assuming the *length* component is long enough to grip the number of the characters. LZW, however, has to construct the pointers step by step, and it will have pointers to two bytes, three bytes etc. Indeed, if the file length is m, we can find the number of the pointers – n, by solving the equation $m=n(n+1)/2$, i.e. $O(\sqrt{n})$ pointers. On the other hand, the pointer of LZSS is more expensive; hence in other cases when the number of the pointers is almost equal, LZW will compress better. This paper shows the extreme case where the number of the pointers is exactly equal; hence the efficiency of LZW is significantly better.

5. REFERENCES

Bell, T.C., Cleary, J.G., &. Witten, I.H. (1990)*"Text Compression,"* Englewood Cliffs, NJ: Prentice-Hall.

Huffman, D.A. (1952) *"*A Method for the Construction of Minimum-Redundancy Codes,*" Proc. IRE*, pp. 1098-1101.

Howard, P.G., Vitter, & J. S. (1992) *"*Analysis of Arithmetic Coding for Data Compression,*"* invited paper in special issue on data compression for images and texts in *Information Processing and Management*, 28(6), 749-763.

Bell, T.C, Witten, I.H, & Cleary, J.G. (1989) "Modeling for Text Compression*," ACM Computing Surveys*, 33(4), 557-591.

Schuegraf, E.J., & Heaps, H. S. (1974) "A Comparison of Algorithms for Data Base Compression by Use of Fragments as Language Elements," *Information Storage and Retrieval* 10, 309-319.

Ziv, J., & Lempel, A. (1977) “A Universal Algorithm for Sequential Data Compression,” *IEEE Trans. on Inform. Theory, IT*-23(3), 337-349.

gzip, Free Software Foundation, Inc. (1991) 675 Mass Ave, Cambridge, MA, USA.

Ziv, J., & Lempel, A. (1978) “Compression of individual sequences via variable-rate coding,*” IEEE Trans*. *Inform. Theory*, 24(5), 530-536.

Willard, L., Lempel, A., Ziv, J. & Cohn, M. (1984) *“Apparatus and method for compressing data signals and restoring the compressed data signals,”* US patent - US4464650.

Storer, J.A., & Szymanski, T.G. (1982) “Data Compression via Textual Substitution,” *Journal of ACM,* 29(4), 928-951.

Welch, T.A. (1984) “A technique for high performance data compression*”*, *IEEE Computer*, 17(6), 8-19.

Bell, T.C., Witten, I.H., & Cleary, J.G. (1988) Modeling for Text Compression, Technical Report 1988-327-39, The University of Calgary, Calgary, Alberta Canada.

Horspool, R.N. (1991) "Improving LZW," Proc. *Data Compression Conference (DCC 91*), Snowbird, Utah, IEEE Computer Society Press, Los Alamitos, CA, pp. 332-341.

Tao, T., & Mukherjee, A. (2004) "LZW Based Compressed Pattern Matching," Proc. *Data Compression Conference* (DCC '04), p. 568.