Improving LZW

R. Nigel Horspool
Dept. of Computer Science, University of Victoria
P.O. Box 3055, Victoria, B.C., Canada V8W 3P6
e-mail address: nigelh@csr.uvic.ca

The Lempel-Ziv-Welch (LZW) compression algorithm is widely used because it achieves an excellent compromise between compression performance and speed of execution. A simple way to improve the compression without significantly degrading its speed is proposed, and experimental data shows that it works in practice. Even better results are achieved with an additional optimization of "phasing in" binary numbers.

1 Introduction

The most effective compression algorithms are, unfortunately, computationally expensive. Unless a special-purpose hardware implementation is available or unless the application has no immediate deadline to meet (as in overnight archiving of files), most users would prefer to trade some compression performance for faster rates of compression and decompression.

At present, the *de facto* method of choice is the Lempel-Ziv-Welch algorithm (LZW) [2]. LZW was originally designed for implementation by special hardware, but it turned out to be highly suitable for efficient software implementations too. An enhanced variant is available on UNIX systems and many other systems as the *compress* command. We refer to this variant as LZC. The speed and compression performance of LZC are a result of careful data structure design for hash table look-ups, some tuning and the addition of logic for restarting the algorithm when the source file changes its characteristics enough to worsen compression performance.

Is it possible to improve the compression performance of LZC even further without significantly affecting its execution speed? To answer this question, we examine some possibilities and select two as having the most promise. The two enhancements together entail only minor changes to the LZC implementation and increase its computational requirements only a little. Experimental results indicate that compression performance is improved, depending on the nature of the source file, by up to 8%. That is, a file that is compressed to 49% of its original size by the standard LZW algorithm is compressed to about 41% of its size by the improved algorithm.

2 Descriptions of LZW and LZC

Ziv and Lempel are the originators of a large and growing family of related compression algorithms [4][5]. Algorithms in this family achieve their compression by replacing a repeated sequence of characters with a reference back to its previous occurrence. The algorithms differ according to how such references are represented and on how to select the sequences that are replaced. Twelve different variations on Ziv-Lempel compression are described in [1]. We now briefly review the LZW algorithm.

LZW is an adaptive technique. As the compression algorithm runs, a changing dictionary of (some of) the strings that have appeared in the text so far is maintained. Because the dictionary is pre-loaded with the 256 different codes that may appear in a byte, it is guaranteed that the entire input source may be converted into a series of dictionary indexes. If α and β are two strings that are held in the dictionary, the character sequence $\alpha\beta$ is converted into the index of α followed by the index of β. A greedy string matching algorithm is used for scanning the input, so if the first character of β is x, then αx cannot be an element of the dictionary. The adaptive nature of the algorithm is due to that fact that αx is automatically added to the dictionary if α is matched but αx is not matched. The algorithm may be expressed as follows. We use the notation `<<s,x>>` to represent the string formed by appending character *x* to the string with number *s* in the dictionary.

```
for( i in the range 0 to 255 )
    add i as a one-character string to the dictionary;
add the empty string λ to the dictionary;
sn = string number of λ;
while( input remains ) {
    read( ch );
    if ( <<sn,ch>> is in the dictionary )
        sn = string number of <<sn,ch>>;
    else {
        write_number( sn );
        if ( dictionary is not full )
            add <<sn,ch>> to next position in dictionary;
        sn = string number of <<ch>>;
    }
}
write_number( sn );
```

The algorithm maintains the prefix property – namely, if αx is a string in the dictionary, then α must be held in the dictionary also. This enables an entry for an *n* character string in the dictionary (n>1) to be encoded as a pair `<<s,x>>` where *s* is the number of the n–1 character prefix and *x* is the final character. The LZC implementation uses hashing to store and to look up the `<<s,x>>` pairs efficiently.

String numbers are output in binary. However, as an optimization, the number of bits used varies according to the dictionary occupancy. If N represents the number of strings currently held in the dictionary, then while N is in the range 256 to 511, each string number is output as a 9-bit binary number. While N is in the range 512 to 1023, string numbers are output as 10-bit numbers, and so on.

The decoding algorithm (executed by the *uncompress* command on UNIX systems) must maintain the same dictionary – adding new strings to the dictionary in a manner that mimics the behaviour of the compression algorithm. We omit a full description of the decoding process, referring the reader instead to [2] or to [1]. Decoding should, of course, execute faster than compression because hash table searching for strings in the dictionary is unnecessary.

When the dictionary has filled up, LZW becomes non-adaptive – it compresses using an unchanging dictionary. For heterogenous files composed of sections with widely differing characteristics (executable files have this nature), the lack of further adaptability may lead to poor overall compression performance. A modification, used in LZC, is to monitor the compression ratio after the dictionary has filled. If the ratio starts to worsen, the dictionary is simply cleared. In effect, the compression algorithm is restarted. A special code (an unused string number) is output so that the decoding algorithm will re-initialize its dictionary at the same point. This enhancement is highly effective on heterogenous files (composed of sections with different characteristics) such as executable files.

3 Possibilities for Improving LZC

Unless we are prepared to completely change the nature of the LZW/LZC algorithm, there appears to be two areas where one might seek better compression performance. One area is concerned with the binary encoding of string numbers (a scheme that uses the same number of bits to represent every string number is surely sub-optimal), and the other area is concerned with the contents of the dictionary. We will look at these two areas in turn.

Redundancy in Encoding String Indexes

Until the dictionary has been filled, a redundant coding for the string numbers is used. If the dictionary currently contains N strings, then $\lceil \log_2(N) \rceil$ bits are used to encode a string number. A fractional number of bits is therefore wasted unless N happens to be a power of 2.Since N increases by one for each string number that is output by the compression number, we can express the total wastage of bits as

$$\sum_{N\,=\,257}^{\max} \lceil \log_2(N) \rceil - \log_2(N)$$

where *max* is the size that the dictionary grows to. There is an upper bound on *max*, normally selected to be a power of two for LZW implementations. If *max* is chosen to be 65536 (the default limit for the *compress* program), and if the source file exactly causes

the dictionary to be filled, the redundancy amounts to 9.9% of the code bytes generated. If further compression proceeds with the dictionary operating at its maximum capacity, this percentage will be reduced somewhat.

What less redundant coding schemes could be used for the string numbers? One possibility is arithmetic coding. If each string number is assigned a probability of 1/*N*, we would eliminate all the wastage. Unfortunately, a software implementation of the arithmetic coding algorithm, as given in [3], would substantially reduce the execution speed of the LZW algorithm. A simpler, and much faster, technique is to phase in binary codes progressively (a scheme described in Appendix A-2 of [1]). An example of a phased-in coding scheme is illustrated in Table 1 for the case when string numbers range over 0 to 9.

Implementation of a phased-in coding scheme is straightforward. For example, the codes shown in Table 1 may be read and converted to integers using the following logic.

```
i = read a 3-bit number;
if (i >= 6) { b = read a 1-bit number; i = 2*i + b; }
```

Conversely, they may be output using the following logic.

```
/* output the code for integer c */
if (c >= 6)
    write c+6 as a 4-bit number;
else
    write c as a 3-bit number;
```

Analysis of the expected number of bits saved with this coding scheme (under the assumption that the string numbers are equally probable) is straightforward and therefore we omit it. Experimental data given in Table 2 show that its use realizes almost all the potential of arithmetic coding. The second row of the table shows the compression that would be achieved with use of arithmetic coding for string numbers; the third row shows the compression using the phased-in string numbering scheme.

There is a second source of redundancy in the coding scheme. Consider the following small example. Suppose that the compression algorithm is processing the input “abcd...” and that it has “abc” and “abca” in its dictionary but does not have “abcd”. In this situation, the number of the string “abc” will be output, and the compression algorithm will begin matching a new string that begins with the letter ‘d’. We observe that the decoding algorithm *knows* the second string does not begin with the letter ‘a’ (otherwise the string “abca” would have been matched). Therefore, the technique used to number strings

Table 1 The Phased-in Binary Coding Scheme (N=10)

Decimal	Binary Code	Decimal	Binary Code
0	000	6	1100
1	001	7	1101
2	010	8	1110
5	101	9	1111

need not allocate any numbers to strings that begin with the letter ‘a’. Allowing for such a possibility is a form of redundancy.

In general, if σ is the string that has just been output, we can determine the set of immediate suffixes as $\{x \mid \sigma x \in \text{table}\}$ and, for encoding the next string, eliminate all strings whose first character is a member of this set from the numbering scheme.

Experiments show that the amount of redundancy introduced by this effect is not very large. Some experimental data is shown in Table 2. The second row of the table shows the compression that would be achieved if arithmetic coding is used for string number encoding (where all string numbers are assumed to be equally probable). The fourth row shows the compression that would be achieved if the impossible strings are eliminated from the numbering system. In other words, the third row corresponds to using arithmetic coding with zero probabilities for the impossible strings and equal probabilities for all other strings. Since the dictionary changes dynamically during much of a typical execution of the LZW algorithm, any coding scheme that takes advantage of this optimization optimization would involve considerable execution overhead. In light of the small potential gain in compression performance, we consider that this direction is not worth pursuing further.

There is another way to view the redundancy in the string number coding scheme. The matching of input against strings in the dictionary uses a *greedy* algorithm. That is, the matching process used in the LZW algorithm goes for the longest match possible. However, alternative (non-greedy) matchings are ignored. For example, if the dictionary contains the four strings “ab”, “abc”, “cd” and “d” (but not the string “abcd”), the sequence of characters “abcd” will be matched as “abc” and then “d”. However, a non-greedy matching of “ab” and “cd” is also available and would be equally acceptable to the decoding algorithm.

Estimating Probabilities for String Numbers

The binary number coding of string numbers standard implementation of LZW has an inherent assumption that all string numbers are equally likely. But that is definitely not the case. As was explained above, there is correlation between consecutive string numbers that limit the possibilities (i.e. some strings have a probability of zero). There are intermediate cases too. If the dictionary contains an entry for string σ, then, as the compression algorithm proceeds, it will add more strings of the form σx. Each addition of one of these

Table 2 Compression with Different String Number Encoding Schemes

	C source file	English Text File
Standard LZW with binary coding	49.3%	36.7%
LZW with arithmetic coding	47.5%	35.4%
LZW with “phased-in” binary coding	47.8%	35.5%
LZW with optimized arithmetic coding	46.9%	34.6%

strings reduces the probability that the string σ needs to be encoded. (If σx is added for every x in the source alphabet, the only remaining circumstance when σ might need to be encoded occurs at the end of the file.)

A potential research direction is to devise a scheme where the LZW implementation makes estimates of probabilities for the string numbers and uses these estimates in arithmetic coding of the string numbers. It is hard to imagine, however, how any such scheme could be implemented efficiently enough to be practical. For that reason, we have not pursued this possibility further.

Possibilities for Adaptive Loading of the Dictionary

The LZW algorithm uses a particularly simple scheme for loading new strings into the dictionary. We can characterize the scheme as follows. If the sequence of strings that are matched is σ_1, σ_2, σ_3, ... then LZW adds new strings of the form

$$\sigma_1 P_1(\sigma_2),\ \sigma_2 P_1(\sigma_3),\ \sigma_3 P_1(\sigma_4),\ ...$$

where $P_k(\sigma_i)$ represents the *k*-character prefix of string σ_i. But it is easy to imagine schemes where strings of the form $\sigma_1\sigma_2$, $\sigma_2\sigma_3$, ... get loaded, or strings of the form $S_1(\sigma_1)\sigma_2$, $S_1(\sigma_2)\sigma_3$, ... where $S_k(\sigma_i)$ represents the *k*-character suffix of string σ_i, or strings of the form $S_1(\sigma_1)\sigma_2 P_1(\sigma_3)$, and so on. The possibilities seem endless. Some possibilities along these lines have been used in Ziv-Lempel variants.

A property that can be used as a criterion for limiting the choice is the prefix property. We can require that if σ is a string in the dictionary then every prefix of σ must also present. Without this property, more sophisticated algorithms for matching the input against the dictionary contents are required and it seems inevitable that there would be a speed penalty. We should also be careful not to load strings into the dictionary at too fast a rate – otherwise the compression and decompression algorithms may spend too much time updating the dictionary (and they will also have to handle dictionary overflow much earlier).

We have a simple proposal for adding potentially useful strings to the dictionary. If the sequence of strings matched by the compression algorithm is σ_1, σ_2, σ_3, ..., as before, then the standard LZW/LZC algorithm will add strings of the form $\sigma_i P_1(\sigma_{i+1})$. We propose adding several strings of the form

$$\sigma_i P_k(\sigma_{i+1}) \qquad \text{for } k = 1, 2, \ldots \min(\text{ maxlen}, |\sigma_{i+1}|\ )$$

where *maxlen* imposes a maximum limit on *k* .

For example, if the input source is the string “abcdefg...” and if LZC matches it as the two strings “abc” and “defg”, LZC will add just “abcd” to its dictionary. We propose also adding “abcde” when *maxlen* Š 2, and adding “abcdef” when *maxlen* Š 3, and so on.

The rationale is that if the substring “abcdefg” occurs a second time in the input, LZC will now add “abcde” to the dictionary. If “abcdefg” occurs for a third time, LZC will next add “abcdef”, and so on. Since the underlying principle of LZW/LZC is to improve the encoding of repeated substrings, our suggestion falls entirely within the spirit of that principle. Furthermore, the prefix property is maintained and so the required modifications to the LZC implementation are minor. The revised algorithm is detailed in the next section. We note that setting a maximum length on the extension (i.e. *maxlen* above) provides a mechanism for varying the rate of loading of new strings into the dictionary. As experimental data will show, increasing the maximum allowed value for *k* improves compression performance but causes the dictionary to fill at a much faster rate.

4 The Accelerated Dictionary Loading Algorithm

The revised compression algorithm that incorporates accelerated loading of strings into the dictionary has the following structure. The variable *xsn* holds the string number of the prefix used in the next extra string to be added to the dictionary. The variable *xscnt* implements control over the number of extra strings generated. The limit on the number of such strings is controlled by the variable *maxlen*.

```
for( i in the range 0 to 255 )
    add i as a one-character string to the dictionary;
add the empty string λ to the dictionary;
sn = string number of λ; xscnt = 0;
while( input remains ) {
    read( ch );
    if ( <<sn,ch>> is in the dictionary ) {
        if ( xscnt > 1) {
            add <<xsn,ch>> to next position in dictionary;
            xsn = string number of <<xsn,ch>>;
            xscnt = xscnt - 1;
        }
        sn = string number of <<sn,ch>>;
    } else {
        write_number( sn );
        if ( dictionary is not full ) {
            add <<sn,ch>> to next position in dictionary;
            xsn = string number of <<sn,ch>>;
            xscnt = maxlen;
        }
        sn = string number of <<ch>>;
    }
}
write_number( sn );
```

It may be observed that if *maxlen* is set to a very large value, exactly one new string is stored in the dictionary for every character read from the input. Note also that the “phased-in” binary coding scheme may, of course, be used for outputting the string numbers.

The decoding algorithm has to add exactly the same set of strings to its dictionary. It seems simplest to create the new strings for dictionary insertion as characters are generated for output. For space reasons, we omit giving details of the algorithm.

5 Experimental Measurements

Some experimental results to show the effect of accelerated dictionary loading on compression performance are given in Table 3. The LZC implementation with accelerated dictionary loading is labelled LZCA in the table; the implementation that uses both accelerated dictionary loading and the "phased-in" numbering scheme is labelled LZCA-P. The parameter that controls the maximum length of the suffix, *maxlen*, is shown across the top of the table. The 'max=1' column corresponds to the standard LZC implementation.

The results show that compression performance tends to improve as strings are added to the dictionary at a faster rate. The amount of improvement depends greatly on the nature of the source file. For files with very little short-range correlations between characters, such as executable files, the improvement is non-existent or negligible. But for ASCII text files containing source code, English text, and the similar, the improvements are significant. As another means of displaying the compression improvement when accelerated dictionary loading is used, Figure 1 shows the overall compression achieved as the source file (an English text file) is processed. The three different curves correspond to loading rates of 1 (i.e. normal LZC), 2 and 5. The curves show that compression with accelerated loading has a consistent advantage over standard LZC. The advantage is only lost when the dictionary is eventually filled.

Table 3 LZW with Accelerated Dictionary Loading

C Source Code File (size 25200 bytes):

Algorithm	**max=1**	**max=2**	**max=3**	**max=4**	**max=5**	**max=∞**
LZWA	49.3%	46.0%	44.7%	44.1%	43.7%	42.8%
LZWA-P	47.8%	44.4%	43.0%	42.3%	42.0%	41.2%

English Text File (size 76816 bytes):

Algorithm	**max=1**	**max=2**	**max=3**	**max=4**	**max=5**	**max=∞**
LZWA	36.7%	34.8%	34.1%	33.8%	33.5%	34.5%
LZWA-P	35.5%	33.5%	32.7%	32.5%	32.2%	32.5%

Sun-3 Executable File (size 24576 bytes):

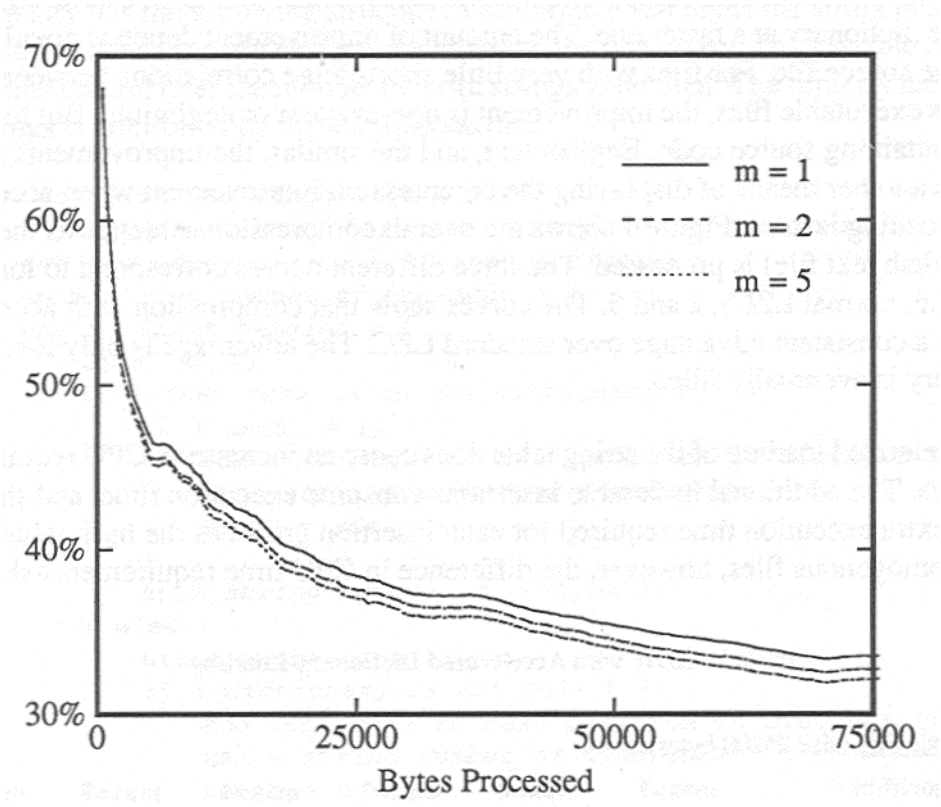
Algorithm	**max=1**	**max=2**	**max=3**	**max=4**	**max=5**	**max=∞**
LZWA	18.4%	18.3%	18.3%	18.4%	18.4%	18.4%
LZWA-P	17.6%	17.5%	17.5%	17.5%	17.5%	17.6%

Accelerated loading of the string table does cause an increase in CPU requirements on short files. The additional hash table insertions consume execution time, and the amount of extra execution time required for each insertion grows as the hash table fills. For large homogenous files, however, the difference in CPU time requirements should be minimal. The standard LZC implementation would simply fill the dictionary and continue to operate with a static dictionary for the remainder of the input. It makes little difference if that dictionary is loaded at a faster rate and becomes static sooner. Some CPU timings on files that does not cause the dictionary to completely fill up are reported in Table 4 below. Timings are measured in CPU seconds (the total of the user time and system time) on a SUN-4 SPARCserver 370 machine.

6 Summary and Conclusions

The better the compression that is achieved by a coding technique, the harder it becomes to extract each percent of additional compression. It has not been easy to find easily implementable methods for improving the performance of LZC, especially when we impose an additional constraint that the execution time requirements should not be severely affected. We have selected two ways of improving LZC (the Unix *compress* command). One, a

Figure 1 Cumulative Compression Rates with Accelerated Loading



method of loading the dictionary at a faster rate, has not been used before. The other, a method to phase in increased lengths of binary numbers gradually, is not original but is not currently used with LZC. Together, these two compression methods achieve substantial improvements, especially on shorter files where the dictionary does not normally have a chance to fill to an extent that achieves good compression performance.

References

[1] Bell, T.G., Cleary, J.G., and Witten, I.H. *Text Compression*. Prentice-Hall, Englewood Cliffs, NJ (1990).

[2] Welch, T.A. “A Technique for High-Performance Data Compression.” IEEE Computer 17,6 (June 1984), pp. 8-19.

[3] Witten, I.H., Neal, R., and Cleary, J.G. “Arithmetic Coding for Data Compression.” Comm. of ACM 30,6 (June 1987), pp. 520-540.

[4] Ziv, J, and Lempel, A. “A Universal Algorithm for Sequential Data Compression.” IEEE Trans. on Inf. Theory IT-23,3 (May 1977), pp. 337-343.

[5] Ziv, J, and Lempel, A. “Compression of Individual Sequences via Variable-Rate Coding.” IEEE Trans. on Inf. Theory IT-24,5 (Sept. 1978), pp. 530-536.

Table 4 CPU Time Measurements

C Source Code File:

max=1	max=2	max=3	max=4	max=5	max=∞
0.23	0.24	0.25	0.26	0.26	0.29

English Text File:

max=1	max=2	max=3	max=4	max=5	max=∞
0.46	0.52	0.57	0.61	0.65	0.83