Practical Fixed Length Lempel Ziv Coding

Shmuel T. Klein[1] and Dana Shapira[2]

[1] Dept. of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel
tomi@cs.biu.ac.il
[2] Dept. of Computer Science, Ashkelon Academic College, Ashkelon 78211, Israel
shapird@ash-college.ac.il

Abstract. We explore the possibility of transforming the standard encodings of the main LZ methods to be of fixed length, without reverting to the originally suggested encodings. This has the advantage of allowing easier processing and giving more robustness against errors, while only marginally reducing the compression efficiency in certain cases.

1 Introduction and Motivation

Some of the most popular compression algorithms are based on the works of J. Ziv and A. Lempel. One of the main features of these methods is their adaptivity: they are dictionary based techniques, in which compression is obtained by replacing parts of the text to be compressed by (shorter) pointers to elements in some dictionary. The ingenious innovation of the LZ methods was to define the dictionary as the text itself, using pointers of the form (offset, length) in the variant known as LZ77 [17], or using a dynamically built dictionary consisting of the strings not encountered so far in the text for the method known as LZ78 [18].

The original Lempel-Ziv algorithms suggested to produce pointers and single characters in strict alternation and thereby enabled the use of a fixed length encoding for the (offset, length, character) items of LZ77 or the (pointer, character) items of LZ78. Later and more efficient implementations then removed the requirement for strict alternation by adding flag-bits, as in [12] for LZ77, or by including the single characters within the dictionary elements, as in [15] for LZ78. This paved the way to using variable length encodings. Indeed, the idea of assigning shorter codewords to items which appear with higher probability has been a main ingredient of some basic compression approaches, such as Huffman coding, which is optimal once the probabilities of the items to be encoded are given, and under the constraint that an integral number of bits should be used for the encoding of each element. But the use of variable length coding has its price:

- The manipulation of variable length codewords, which are thus not always byte aligned, is much more involved, requiring algorithms that are more complicated and usually also more time consuming. While increased time might be acceptable in certain applications for the encoding, the *decoding* is often required to be very fast, and fixed length codes can contribute to this goal.
- The compressed text is more vulnerable to transmission errors.

Another disadvantage of variable length codes is that direct search in the compressed file is not always possible, and even if it is, it might be slower than in the case of fixed length encodings. In the case of LZ schemes, even the processing of fixed length encodings might be problematic, as will be explained below.

These deficiencies of variable length codes have led recently to the investigation of several alternatives. It was first suggested to trade the optimal compression of binary Huffman codes with a faster to process 256-ary variant. The resulting codewords have lengths which are multiples of 8, so consist of 1, 2 or more bytes, and for large enough alphabets, the loss in compression efficiency is of the order of only a few percent [4], and may often be tolerated. The problem of getting large enough alphabets for the method to be worthwhile was overcome, on natural text files, by defining the elements of the "alphabet" to be encoded as the different *words* of the database, instead of just the different characters [11].

A further step was then to use a *fixed* set of codewords, rather than one which has to be derived according to the given probability distribution as for Huffman codes, still adhering to the main idea of variable length codes with codeword lengths that are multiples of bytes [3,2]. Another tradeoff with better compression but slower processing can be obtained by the use of Fibonacci codes [6]. Both (s, c) codes and Fibonacci codes have the additional advantage of lending themselves to searches directly in the compressed file. Finally, at the other end of the spectrum, one may advocate again the use of fixed length codes, achieving compression by using variable length for the strings to be encoded instead for the codewords themselves [13,9].

We now turn to the investigation of how to adapt Lempel and Ziv codes to fit into a framework of fixed length codes without reverting to the original encodings based on alternating pointers and characters, yielding some of the above mentioned advantages. The new variants are presented in the next section, and some experimental results are given in Section 3.

2 Fixed length LZ codes

The two main approaches suggested by Lempel and Ziv have generated a myriad of variants. We shall more specifically refer to LZSS [12] as representative of the LZ77 family, and to LZW [15] for LZ78.

2.1 Fixed length LZSS

LZ77 produces an output consisting of a strictly alternating sequence of single characters and pointers, but no external dictionary is involved and the pointers, coded as (offset, length) pairs, refer to the previously scanned text itself. LZSS suggests to replace strict alternation by a set of flag-bits indicating whether the next element is a single character or an (offset, length) pair. A simple implementation, like in LZRW1 [16], uses 8 bits to encode a character, 12 bits for the offset and 4 bits for the length. Excluding the flag-bits, we thus get codewords of lengths 8 or 16. Though variable length, it still keeps byte alignment, by processing the flag bits by blocks of 16.

One of the challenges of LZ77 is the way to locate the longest matching previously occurring substring. Various techniques have been suggested, approximating the requested longest match by means of trees or hashing as in [16]. In many cases, fixed length codes are preferred in a setting in which encoding and decoding are not considered to be symmetric tasks: encoding might be done only once, e.g., during the construction of a large textual database, so the time spent on compression is not really relevant. On the other hand, decompression is requested every time the database is being accessed and ought therefore to be extremely fast. For such cases, there is no need to use hashing or other fast approximations for finding the longest possible match, and one might find the true optimum by exhaustive search. Given the limit on the encoding of the offset (12 bits), the size of the window to be scanned is just 4K, but the scan has to be done for each position in the text.

Encoding and decoding A fast way to get a fixed length encoding is to set the length of the shortest character sequence to be copied to 3 (rather than 2 in the original algorithm); thus when looking for the longest previously occurring sequence P which matches the sequence of characters C starting at the current point in the text, if the length of P is less than 3, we encode the first 2 characters of C. In the original algorithm, if the length of P was less than 2, we encoded only the first character of C. The resulting encoding therefore uses only 16 bits elements, regardless of if they represent character pairs or (offset, length) pairs. To efficiently deal also with the flag bits, one can use the same technique as in [16], aggregating them into blocks of 16 elements each.

Decoding could then be done by the following simple procedure:

```
$j \longleftarrow 0$
while not EOF
    $F[0 \cdot\cdot 15] \longleftarrow$ next 2 bytes
    for $i \longleftarrow 0$ to 15
        if $F[i] = 0$ then
            $(T[j], T[j+1]) \longleftarrow$ next 2 bytes
            $j \longleftarrow j+2$
        else
            $z \longleftarrow$ next 2 bytes
            off $\longleftarrow 1 + z \bmod 2^{12}$
            len $\longleftarrow 3 + \lfloor z/2^{12} \rfloor$
            $T[j \cdot\cdot j+\text{len}-1] \longleftarrow T[j-\text{off} \cdot\cdot j-\text{off}+\text{len}-1]$
            $j \longleftarrow j+\text{len}$
    end for
end while
```

Compression efficiency At first sight, enforcing fixed length codewords seems quite wasteful with the LZRW1 encoding scheme:

1. The number of characters that are not encoded as part of a pointer will increase, so the average length of a substring represented by a codeword, and thus the compression, will decrease. Moreover, each such character actually adds a negative

contribution to the compression, because the need of the flag bit: a single character is encoded by 9 bits in the original LZSS and by 8.5 bits in the fixed length variant, whereas only 8 bits are needed in the uncompressed file.

2. Elements of the form (off,2) are replaced by a pair of characters, and both are encoded by the same number of bits.
3. If there is a sequence of odd length ℓ of characters encoded on their own in LZSS, followed by a pointer (off, len), with len > 2, one would need $9\ell + 17$ bits to encode them. This can be replaced in the fixed length variant by $(\ell+1)/2$ pairs of characters, followed by the pointer (off, len-1), requiring $17\left(\frac{\ell+1}{2}+1\right)$ bits. There is thus a loss whenever $\ell < 17$, and this will mostly be the case, as the probability of having sequences of single characters of length longer than 17 in the compressed text is extremely low: it would mean that the sequence contains no substring of more than two characters which occurred earlier. In all our experiments, the longest sequence of single characters was of length $\ell = 14$.

On the other hand, the passage to fixed length encoding can even lead to additional savings in certain cases. An extreme example would be an LZSS encoded file in which all the sequences of single characters are of even length. In the corresponding fixed length encoding, all the (off, len) pairs would be kept, and the single characters could be partitioned into pairs, each requiring only 17 bits for its encoding instead of 18 bits if encoded as two 9-bit characters. But the improvement can also originate from a different parsing, as can be seen in the example in Figure 1.

File	Text	Size
Original	b c d e f b b c d a b b c d e f b b	18 bytes
variable LZSS	b c d e f b (6,3) a (5,4) (11,4)	7 chars + 3 pointers + 10 bits $\rightarrow$ 15 bytes
fixed length	b c d e f b (6,3) a b (5,3) (11,4)	4 pairs + 3 pointers + 7 bits $\longrightarrow$ 15 bytes
better fixed	b c d e f b (6,3) a b (11,7)	4 pairs + 2 pointers + 6 bits $\longrightarrow$ 13 bytes

Figure 1. Comparison of LZSS encodings

The first line brings a small example of a text of 18 characters and the next line is its LZSS encoding using the LZRW1 scheme: it parses the text into 10 elements, 7 single characters and 3 (off, len) pointers, for a total of $7 \times 1 + 3 \times 2 + 10$ flag bits (rounded up to 2 bytes) = 15 bytes. The third line shows a parsing which could be derived from the one above it, but does not encode single characters, so that a (5,4) has to be replaced by a b (5,3). There are now 11 elements in the parsing, 4 character pairs and 3 pointers, so they require already 14 bytes, to which one has to add the flags. The last line shows that in this case, another parsing is possible, using only 6 codewords, which yields 12 bytes plus the flag bits. This example shows that one can get a strict improvement with fixed length encoding.

An encoding as in LZRW1 is of course not the only possibility for fixed lengths, but the 16 bit units have been chosen because it is most convenient to process the input by multiples of bytes. If one is willing to abandon byte alignment, but still wants to keep fixed length, one could for example define codewords of 18 bits and incorporate the flag bits into the codewords themselves. The processing could be done by blocks of 18 bytes = 144 bits, each block representing 8 consecutive 18 bit codewords. One could then use 13 bits for offsets of size 1 to 8192, and stay with

4 bits for copy lengths between 3 and 18. Alternatively, the 18 bit codewords could be stored as before in units of 2 bytes, considering the exceeding 2 bits as flag-bits, which could be blocked for each set of 8 consecutive codewords.

For a further refinement, note that only 17 of the 18 bits are used in case of a character pair. For example, the second bit from the left could be unused. To fully exploit the encoding possibilities, decide that this second bit will be 0 when the following 16 bits represent a character pair; this frees 16 bits in the case the second bit is set to 1. One can then use these 16 bits for (off, len) pairs as before in the 16-bit encoding for offsets from 1 to 4096, and shift the range of the offsets encoded by the 18-bit codewords accordingly to represent numbers between 4097 and 12288. The increased size of the search window will lead to improved compression.

The above methods are suitable for alphabets with up to 256 symbols. For the general case of a byte aligned fixed length encoding of LZSS, consider an alphabet Σ, of size $|\Sigma|$, and denote the size of a byte by B. We shall adapt the encoding algorithm, the size W of the sliding window and the maximum copy length L as follows. If $\log(|\Sigma|) = kB$ for some integer k, choose W and L such that

$$\log W + \log L = kB, \tag{1}$$

and use throughout codewords of fixed length k bytes. If $\log(|\Sigma|)$ is not a multiple of B, let $k = \lceil \log(|\Sigma|)/B \rceil$ and again use codewords of k bytes, only that in this case, a codeword for a single character has r spare bits, $1 \le r < B$, which we shall use to accommodate the required flagbits. The codewords for copy elements are still defined by equation (1).

The first element does not need a flag, as it must be a single character. The r flags in codewords representing single characters refer to the *following* r items that have not yet been assigned a flag. For example, if the second and third elements in the compressed file are single characters, we have already collected $2r$ flag bits, referring to the elements indexed $2, 3, \ldots, 2r + 1$. If at some stage we run out of flag bits, a codeword consisting only of kB flags is inserted. This may happen in the above example if the elements indexed $3, \ldots, 2r + 1$ are all of type (`offset, length`). If there are many items of type single character, this scheme might be wasteful, as a part of the available flag bits will be superfluous, but in a typical scenario, after some initialization phase, an LZSS encoded file consists almost exclusively of a sequence of copy items, with only occasionally interspersed single characters.

Robustness A major reason for the vulnerability of variable length codes to transmission errors — a bit flip from 0 to 1 or 1 to 0, a bit getting lost or a spurious bit being picked up — is that the error might not be locally restricted. If one considers only changes in bit values, but assumes that there are no bit insertions or deletions, then in the case of fixed length codes, only a single codeword will be affected. But for variable length codes, the bit flip might imply that the encoded text will now be parsed into codewords of different lengths, and the error can then propagate indefinitely. Consider, for example, the code $\{01, 10, 000, 001, 110, 111\}$ for the alphabet {A, B, C, D, E, F}, respectively, and suppose the encoded text is EBBB$\cdots$BA = 110101010$\cdots$1001. If the leftmost bit is flipped, the text will erroneously be decoded as AAAA$\cdots$AD, which shows that a single bit error can affect an unlimited number of consecutive codewords.

For the case in which the number of bits can also change, fixed length codes might be the most vulnerable of all. An inserted or deleted bit will cause a shift in the decoding, and all the codeword boundaries after the error could be missed. In variable length encoded files, on the other hand, the error might also spill over to a few consecutive other codewords, but there is always also a chance that synchronization will be regained. This actually happens often after a quite low number of codewords for Huffman codes [8], a property which has several applications, see [10].

As to LZSS encoded texts, bit insertions or deletions are hazardous for both the fixed and the original variable length variants. For bit flips, if they occur in the encoding of a character, it will be changed into another one, so the error will be local; if the bit flip occurs in an offset item, a wrong sequence will be copied, but again, only a restricted range will (generally) be affected; if the error is in a length item, a wrong number of characters will be copied (yet starting from the correct location), which is equivalent to inserting or deleting a few characters; if one of the flag bits is flipped, then the text encoded by the original LZSS could be completely garbelled, while for the fixed length variant, no harm is done other than wrongly decoding a codeword of one type as if it were of the other, since both types are encoded by 16 bits.

There might be the danger of an unbounded propagation of the error even in the case of a bit flip in an offset or length item: the wrong characters could be referenced later by subsequent (off, len) pairs, which would again cause the insertion of wrong characters, etc. Such a chain of dependent errors is not unlikely in natural language texts: a rare word might be locally very frequent, e.g., some proper name, and each occurrence could be encoded by a reference to the preceding one. If there is an error in the first occurrence, all the subsequent ones might be affected.

In fact, LZSS encodings are error prone on two accounts: because of the variable lengths, (1) the parsing into codewords could be wrong, but in addition, even if the parsing and thus the codewords are correct, (2) their interpretation might suffer when previous errors caused an erroneous reconstitution of the referenced text. Fixed length LZSS is robust against the first type of error, but not against the second.

Compressed Pattern Matching Given a pattern of size m and a compressed file of size n, *Compressed Pattern Matching* is the problem of locating a pattern directly in the compressed file without any decompression, in time proportional to the size of the input. Performing pattern matching in LZSS compressed files is not trivial, as the encoding of the same subpattern is not necessarily the same throughout the file and depends also on its location. A variant of LZSS that is suitable for compressed matching, replacing the backward by forward pointing copy elements, has been suggested in [7], but the problems remain essentially the same for fixed length as for variable length encodings.

2.2 Fixed length LZW

LZW is based on parsing the text into phrases belonging to a dictionary, which is dynamically built during the parsing process itself. The output of LZW consists of a sequence of pointers, and the size of the encoding of the pointers is usually adapted to the growing dictionary: starting with a dictionary of 512 entries (256 for a basic

ASCII alphabet as the single characters have to be included, and room for 256 additional items), one uses 9-bit pointers to refer to its elements, until the dictionary fills up. At that stage, the number of potential entries is doubled and the length of the pointers is increased by 1. This procedure is repeated until the size of the dictionary reaches some predetermined limit, say 64K with 16-bit pointers. In principle, the dictionary could be extended even further, but erasing it and starting over from scratch has the advantage of allowing improved adaptation to dynamically changing texts, while only marginally hurting the compression efficiency on many typical texts.

Encoding and decoding A fixed length encoding variant of LZW could thus fix the size S of the dictionary in advance and use throughout $\log S$ bits for each of the pointers. This would have almost no effect on the encoding and decoding procedures, besides that there is no need to keep track of the number of encoded elements, since their size does not change.

Compression efficiency The exact loss incurred by passing from the variable to fixed length LZW can be evaluated based on the fact that after each substring parsed from the text, a new element is adjoined to the dictionary. Suppose we let the dictionary grow up to a size of 2^k entries. For the first 256 elements of the encoded text, the loss of using fixed instead of variable length is $256(k-9)$ bits; for the next 512 elements, the loss is $512(k-10)$, etc. The penultimate block has 2^{k-2} elements, for each of which only 1 bit is lost, and in the last block, of 2^{k-1} elements, there is no loss as the maximum of k bits are needed anyway. The total loss in bits is thus

$$\sum_{i=1}^{k-9} 2^{k-i-1} i = 2^k - (k-7)2^8. \tag{2}$$

The size of the fixed length compressed file at this stage is k times the total number of encoded elements,

$$k \sum_{i=8}^{k-1} 2^i = k(2^k - 2^8). \tag{3}$$

The relative loss, expressed as a fraction of the file size, is then the ratio of (2) to (3), which is plotted in Figure 2. Typical values are about 6.05% for $k = 16$ and 4.17% for $k = 24$, and in any case the loss does not exceed 6.463% (for $k = 14$).

For larger files, if the dictionary is rebuilt each time it reaches a size of 2^k elements, one may consider it as a sequence of blocks, each of which, except perhaps the last, representing the encoding of 2^k elements. The overall relative loss would thus approach the values above for large enough files. Another option, which can also be useful for the regular LZW, is to consider the dictionary as constant once it fills up. This approach is actually a good one when the file is homogeneous, like, e.g., some natural language text. In that case, the relative loss of using fixed length codewords already from the beginning, and not only after 2^{k-1} elements have been processed, will decrease to zero with increasing file size.

Robustness From the point of view of robustness to errors, fixed and variable length LZW are identical. Note that a bit flip cannot change the length of one of the codewords, even if those are of variable length, because the length is determined by an

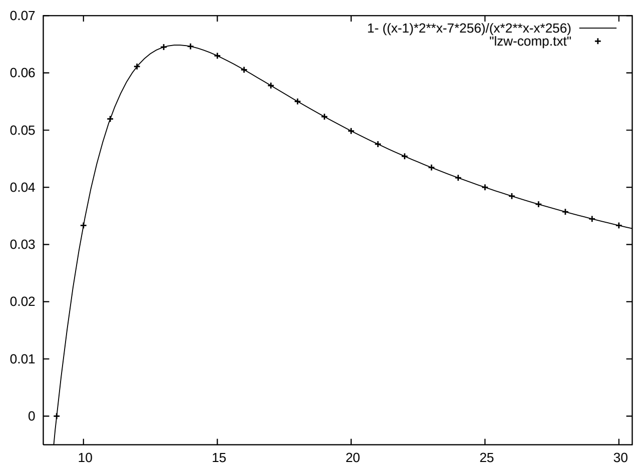

Figure 2. Fraction of loss by using fixed length LZW as function of codeword length

external mechanism (the number of elements processed so far) and does not depend on the value of the codeword itself. Therefore a bit flip will not affect subsequent codewords, and the error seems at first sight to be locally restricted. There might, however, be a snowball effect in case the error occurs during the construction of the dictionary: a wrong bit implies an erroneously decoded codeword, which may cause one or more wrong codewords to be inserted into the dictionary; if these are later referenced, they will cause even more such wrong insertions, and in the long run, the text may become completely garbelled.

If the dictionary is erased when it reaches 2^k entries, such errors cannot propagate beyond these new initialization points. On the other hand, if the dictionary is considered as fixed once it reaches it limiting size, a bit flip in the construction phase can cause considerable damage, but a bit error occurring later will only destroy a single codeword.

Compressed Pattern Matching Amir, Benson and Farach [1], were the first to perform Compressed Pattern Matching in LZW. After preprocessing the pattern, a so-called LZW trie is built, and used to check at each stage whether the pattern has been found. Since the algorithm requires the extraction of all the codewords for building the trie, the difference between fixed and variable length encodings depends on the number of accesses to the encoded file. One should distinguish between the efficient byte aligned operations and the more expensive operations requiring bit manipulations. If the size of the dictionary is 2^{8k} for some integer k, each codeword of the fixed length LZW encoding requires a single byte oriented operation (by fetching k bytes at a time). However, the codewords of the variable length LZW encoding require bit manipulations in addition to byte extractions. Although the size of the compressed file in fixed length LZW is larger than for variable length LZW, the compressed matching algorithm depends on the number of codewords, which is identical in the two schemes. Therefore, compressed pattern matching in fixed length encoding is less time consuming than pattern matching in variable length LZW.

Figure 3 gives a schematic view of the layout of LZW codewords in the case of variable length. The lower part (b) shows the increase of the codeword sizes from left to right, and the upper part (a) zooms in on the subfile in which only 10-bit codewords are used. Solid lines refer to byte boundaries and the broken lines to codeword boundaries. The appearance of a broken line which does not overlap with a solid line indicates that bit manipulations are required. In the case of fixed length encodings, the codeword length can be chosen so as to minimize the bit specific operations, whereas for the variable length encodings, the non-synchronization of the byte and codeword boundaries can not be avoided.

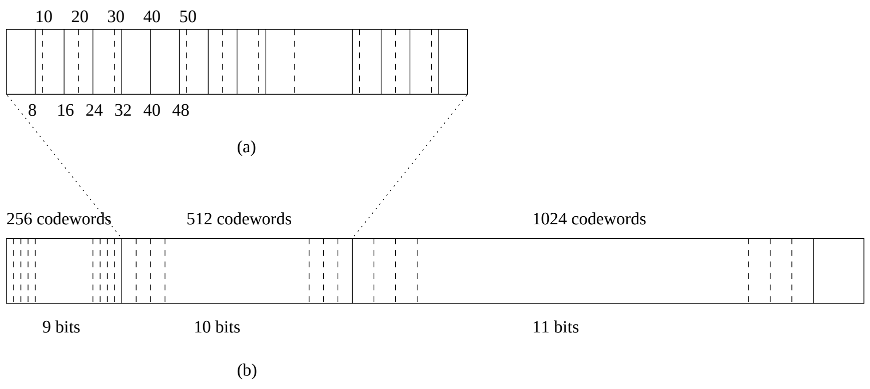

Figure 3. Variable length LZW encoding

3 Experimental results

To empirically compare the compression efficiency, we chose the following input files of different sizes and languages: the Bible (King James version) in English, the French version of the European Union's JOC corpus, a collection of pairs of questions and answers on various topics used in the ARCADE evaluation project [14], and the concatenated text of all the XML files of the INEX database [5]. To get also alphabets of different sizes, the Bible text was stripped of all punctuation signs, whereas the French text and the XML file have not been altered.

File	Size (MB)	gzip	bzip
Bible	2.96	0.279	0.205
JOC corpus	7.26	0.306	0.212
XML	494.7	0.278	0.202

Table 1. Test file statistics

Table 1 brings basic statistics on the files, their sizes in MB and some measure of compressibility, in terms of bzip2 and gzip compression. All compression figures are given as the ratio between the size of the compressed file to that of the uncompressed one. Table 2 deals with LZSS and brings results for 16 and 18 bit codewords. The columns headed hash refer to the approximate method of [16], in which the matching substring is located by hashing character pairs. In our case, we used a full table of 2^{16} entries for each possible character pair; when a previous occurrence was found, it was extended as much as possible. Much better compression could be obtained by using an optimal variant, searching the full addressable window for the longest match, the column headed fix referring to the fixed length variant, and the column headed var to the original one using variable length. The column entitled loss shows the relative loss in percent when using fixed length LZSS, which can be seen to be low.

LZSS	16 bit				18 bit			
	hash	fix	var	loss	hash	fix	var	loss
Bible	0.664	0.398	0.398	0.05%	0.694	0.345	0.331	4.0%
JOC corpus	0.732	0.452	0.451	0.42%	0.760	0.388	0.372	4.1%
XML	0.637	0.412	0.409	0.65%	0.655	0.357	0.340	4.8%

Table 2. Comparing fixed with variable length compression for LZSS

Table 3 is then the corresponding table for LZW, with dictionaries addressed by pointers of 12, 16 and 18 bits. The test files being homogeneous, we used the variant building the dictionary until it fills up, and keeping it constant thereafter. This explains why the loss incurred by using fixed length is decreasing with the size of the input file.

LZW	12 bit			16 bit			18 bit		
	fix	var	loss	fix	var	loss	fix	var	loss
Bible	0.444	0.444	0.02%	0.341	0.339	0.75%	0.313	0.303	3.35%
JOC corpus	0.482	0.482	0.009%	0.346	0.345	0.30%	0.306	0.302	1.38%
XML	0.605	0.605	0.001%	0.484	0.484	0.004%	0.436	0.436	0.01%

Table 3. Comparing fixed with variable length compression for LZW

4 Conclusion

We saw that there is only a small increase in the size of the compressed file when passing from the standard variable length LZ encodings to fixed length variants. In many applications, it may thus be worthwhile to consider this option, which gives some known advantages, like more robustness and easier processing.

References

1. A. Amir, G. Benson, and M. Farach: *Let sleeping files lie: Pattern matching in Z-compressed files*. Journal of Computer and System Sciences, 52 1996, pp. 299–307.
2. N. R. Brisaboa, A. Fariña, G. Navarro, and M. F. Esteller: *(S,C)-dense coding: an optimized compression code for natural language text databases*, in Proc. Symposium on String Processing and Information Retrieval SPIRE'03, vol. 2857 of Lecture Notes in Computer Science, Springer-Verlag, 2003, pp. 122–136.
3. N. R. Brisaboa, E. L. Iglesias, G. Navarro, and J. R. Paramá: *An efficient compression code for text databases.* Proc. European Conference on Information Retrieval ECIR'03, 2633 2003, pp. 468–481.
4. E. S. de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates: *Fast and flexible word searching on compressed text.* ACM Trans. on Information Systems, 18 2000, pp. 113–139.
5. G. Kazai, N. Gövert, M. Lalmas, and N. Fuhr: *The INEX evaluation initiative*, in Intelligent search on XML data, LNCS 2818, Springer-Verlag, 2003, pp. 279–293.
6. S. T. Klein and M. Kopel Ben-Nissan: *Using Fibonacci compression codes as alternatives to dense codes.* Proc. Data Compression Conference DCC–2008, 2008, pp. 472–481.

7. S. T. Klein and D. Shapira: *A new compression method for compressed matching.* Proc. Data Compression Conference DCC–2000, 2000, pp. 400–409.
8. S. T. Klein and D. Shapira: *Pattern matching in Huffman encoded texts.* Information Processing & Management, 41(4) 2005, pp. 829–841.
9. S. T. Klein and D. Shapira: *Improved variable-to-fixed length codes*, in Proc. Symposium on String Processing and Information Retrieval SPIRE'08, Lecture Notes in Computer Science, Melbourne, 2008, Springer-Verlag, pp. 39–50.
10. S. T. Klein and Y. Wiseman: *Parallel Huffman decoding with applications to JPEG files.* The Computer Journal, 46(5) 2003, pp. 487–497.
11. A. Moffat: *Word-based text compression.* Software — Practice & Experience, 19 1989, pp. 185–198.
12. J. A. Storer and T. G. Szymanski: *Data compression via textual substitution.* Journal of the ACM, 29(4) 1982, pp. 928–951.
13. B. P. Tunstall: *Synthesis of noiseless compression codes*, PhD thesis, Georgia Institute of Technology, Atlanta, GA, 1967.
14. J. Véronis and P. Langlais: *Evaluation of parallel text alignment systems: The arcade project.* Parallel Text Processing, J. Véronis, ed., 2000, pp. 369–388.
15. T. A. Welch: *A technique for high-performance data compression.* IEEE Computer, 17 1984, pp. 8–19.
16. R. N. Williams: *An extremely fast Ziv-Lempel data compression algorithm.* Proc. Data Compression Conference DCC–91, 1991, pp. 362–371.
17. J. Ziv and A. Lempel: *A universal algorithm for sequential data compression.* IEEE Trans. on Information Theory, 23 1977, pp. 337–343.
18. J. Ziv and A. Lempel: *Compression of individual sequence via variable rate coding.* IEEE Trans. on Information Theory, 24 1978, pp. 530–536.