Enhancements to Ziv-Lempel Data Compression*

by

Clyde Rogers
Clark D. Thomborson (a.k.a. Thompson)†

January 11, 1989

Technical Report 89-2

University of Minnesota
Duluth

Computer Science

Enhancements to Ziv-Lempel Data Compression*

by

Clyde Rogers
Clark D. Thomborson (a.k.a. Thompson)†

January 11, 1989

Technical Report 89-2

Department of Computer Science
University of Minnesota
Duluth, Minnesota 55812
U.S.A.

*Submitted to the Proceedings of the Computers and Mathematics 1989 Conference, Springer-Verlag, to be held June 12-16 at MIT.

†Supported in part by the National Science Foundation, through its Design, Tools and Test Program under grant number MIP 8706139.

Abstract

We describe several modifications to the Ziv-Lempel data compression scheme, improving its compression ratio at a moderate cost in run time. Our best algorithm reduces the length of a typical compressed text file by about 25 percent. The enhanced coder compresses approximately 1800 bytes of text every second before optimization, making it fast enough for regular use.

1 Introduction

This paper describes several modifications to the Ziv-Lempel data compression algorithm that can reduce the size of output files significantly. Common Ziv-Lempel implementations, such as UNIX compress, reduce the size of our English text and source code test files by 46 to 70 percent. Our enhancements improve this compression ratio to between 55% and 84%, making typical output files about 25 percent smaller.

Cleary and Witten [1] have used Markov modeling, building huge data structures, to achieve somewhat better compression. After embarking on this research, we learned of experiments run by Miller and Wegman [3] on a different set of Ziv-Lempel enhancements.

Section 2 describes the Ziv-Lempel (ZL) and arithmetic coding algorithms. Section 3 describes the combination and extension of these powerful coding techniques, and section 4 describes experiments on the resulting enhanced coder. Section 5 discusses the results of the experiments and points out future directions in Ziv-Lempel coders.

2 Data Compression

Most of today's data compression techniques employ a collection of strings called a dictionary. Any text you might want to compress can be represented by an ordered concatenation of some (or all) of the dictionary's strings. For example, if a dictionary contains all lower case letters, the word "stuff" is represented by the strings "s", "t", "u", "f" and "f". If we assume that the dictionary contains all lower case letters and all possible combinations of two lower case letters, the word "stuff" could be represented by the strings "st", "uf" and "f", the strings "st", "u" and "ff", or one of many other concatenations.

This example shows one way data compression can occur. With the first dictionary, we were able to represent the word "stuff" using five dictionary entries. The second dictionary required only three entries to represent the same word, giving us forty percent compression.

We could stop and congratulate ourselves with this accomplishment, but first we must look a little deeper into what we've done. If our dictionary contains twenty-six entries, as the first dictionary did, we can represent each entry with a five bit number. Using five bits to represent each string, it would encode the word "stuff" in twenty-five bits. Our second dictionary, however, contains 702 entries, so representing each entry takes ten bits. Then the three entry representation of the word "stuff" takes thirty bits. Good thing we didn't congratulate ourselves too soon.

Now consider a third dictionary that contains all lower case letters and the string "st". We could represent the word stuff as "st", "u", "f" and "f", and we

can represent each of the twenty-seven dictionary entries with a five bit number, so this four entry representation takes twenty bits. Finally, we have compressed the word "stuff" by twenty percent from the five entry version.

This example illustrates the two parts of an effective compression technique. First, we need a dictionary that contains more than just strings, it should contain only useful strings. Second, we need to represent those strings in as few bits as possible.

2.1 Ziv-Lempel-Welch Coding

One data compression technique people use daily is to replace regularly used long words with short words. For example, people typically replace the word "automobile" with the word "car". This eliminates seven of ten letters, giving us seventy percent compression without the aid of a computer. Why did we make the shorter word? Because we used the long word often enough that our tongues began to trip over it. So how can a computer know when to make a shorter word? Further, how can the computer know what words to shorten before it looks at the text it must compress?

Using the Ziv-Lempel data compression algorithm [7] [8] [9], an industrious person (or a lazy computer) can mechanically check a text while compressing it, identify strings that are used often and put them into its dictionary. This may seem like a difficult task, but the following example shows the simple ideas behind a variation of the Ziv-Lempel algorithm called the Ziv-Lempel-Welch (ZLW) [5] algorithm.

A ZLW coder begins by seeding its dictionary with all possible one character strings. For example, if a ZLW coder is designed to compress files of eight-bit text, it would start by seeding its dictionary with all 256 eight-bit characters.

ZLW coding looks at the first uncoded character from the source text, and finds the longest string in the dictionary that matches that character and its successors. The coder emits the code for that string, and inserts one new code into its dictionary —a code for that string concatenated with the character that follows it in the source text. The ZLW coder then repeats this process, starting with the next uncoded character, until the source text ends.

Figure 1 illustrates this process using the word "Mississippi" as source text.

Notice that ZLW coding does not start by creating all two letter combinations, then all three letter combinations and so on. It creates new dictionary entries based on what it has already seen in the source text. It assumes that what it sees once in a document it's bound to see again. If this assumption holds true, ZLW coding gives excellent compression.

The ZLW algorithm results in a fair distribution of characters between the source and dictionary —characters that appear often in the source quite naturally appear often in the dictionary. Additionally, as will be demonstrated later

Step	Next Uncompressed Character	Longest Matching String	Emit Code for	Insert String into Dictionary
1	M	M	M	Mi
2	i	i	i	is
3	s	s	s	ss
4	s	s	s	si
5	i	is	is	iss
6	s	si	si	sip
7	p	p	p	pp
8	p	p	p	pi
9	i	i	i	$\emptyset$

Table 1: ZLW Compression Algorithm Example

in this paper, ZLW coding adapts well to repeated character strings.

Because ZLW coding is a mechanical process, reversing it is straightforward. Figure 2 shows the ZLW decoding algorithm run on on an encoding of the word "Mississippi".

Step	Read Code For	Change Last Inserted Code to	Emit Letters	Insert String into Dictionary
1	M	$\emptyset$	M	M?
2	i	Mi	i	i?
3	s	is	s	s?
4	s	ss	s	s?
5	i	si	is	is?
6	s	iss	si	si?
7	p	sip	p	p?
8	p	pp	p	p?
9	i	pi	i	i?

Table 2: ZLW Decompression Algorithm Example

The biggest drawbacks to ZLW coding are that although it creates many useful dictionary entries, it also creates many useless entries. It also adapts too slowly to highly repetitive text files.

2.2 Arithmetic Coding

Back when we encoded the word "stuff", we used a fixed number of bits for each code emission. Given a dictionary of N entries, we represented each entry with

a $\lceil \log N \rceil$-bit number. Many modern coders don't use this system, but instead use Huffman coding [2].

Huffman coding can represent dictionary entries with different numbers of bits based on a probability distribution. Even if you want all characters to have the same probability, when $\log N$ isn't an integer Huffman coding will use fewer than $\lceil \log N \rceil$ bits for some characters. Huffman coding, however, does use an integral number of bits for each dictionary entry. This is fine if all probabilities are powers of two, but won't give optimal results otherwise. For example, given two entries "a" and "b", where "a" has a probability of 0.99, and "b" has a probability of 0.01, you still must represent each entry with a one bit code.

Arithmetic coding [6], on the other hand, allows output of partial bits. If a code should be represented by 1/100 of a bit, arithmetic coding will represent it with 1/100 of a bit.

For example, let's encode the string "aaabbbc", where "a" and "b" each have a probability of 0.3, and "c" has a probability of 0.4. Huffman coding will assign one bit to "c", and two bits each to "a" and "b", so the output will be 13 bits long. Arithmetic coding will assign $-\log(0.3) \cong 1.74$ bits to each "a" and "b", and $-\log(0.4) \cong 1.32$ bits to "c", so the output will be about 11.76 bits long.

So, the chief advantage of arithmetic coding over Huffman coding is that it can emit a code using fractions of bits. The biggest disadvantage of arithmetic coding is that it runs slower than Huffman coding.

3 The ZLWAx Coders

The ZLWAx family of data compressors build their dictionaries using the Ziv-Lempel-Welch algorithm and represent dictionary entries using arithmetic coding. Members of the ZLWAx family employ a predictive strategy that takes advantage of the arithmetic coder's ability to assign different probabilities to different dictionary entries, and two enhancements to the basic ZLW algorithm.

3.1 Predictive Strategy (ZLWAP)

The predictive strategy tries to exploit the idea of context in the source text. A context is a part of a source text that is unlike other parts. For example, if a source contains text and a bit map, the text could be one context and the bit map another. A source text can contain any number of contexts.

The predictive strategy used in the ZLWAP coder is based on two assumptions: first, certain entries are used often when the source file's context changes, and second, new entries from the current context are likely to be useful.

To identify dictionary entries that are used to build up contexts, the coder keeps track of how often older entries are used. If an entry has been around a

long time, and is used regularly, it is likely that it is a building block for new contexts. To give these entries the probability they deserve, the coder gives older dictionary entries probability based on how often they are used. Taking an analogy from the job market, the coder gives probability to older codes based on their experience.

New entries, on the other hand, won't have much experience. The ZLWAP coder, expecting great things from these entries, gives them probability based on their potential.

To differentiate the new entries from old entries, the dictionary for a ZLWAP coder is ordered from its newest entry to its oldest entry. Floating underneath this ordered dictionary are *bins*. Bins break the entries into groups where each entry in a bin has the same probability. So bins near the dictionary front have high probability because they contain new codes, and bins near the dictionary back get probability based on how often the entries they contain are emitted. Bins have two functions: first, they level out the differences between adjacent entries, and second, they make the ZLWAx implementation run much faster. Data describing bins is given in figure 1.

Information relevant to figuring out bin probabilities is given in figure 2. We ran a series of experiments, to be described in Clyde Rogers' Master's thesis, to determine optimal default settings for the parameters τ, S, *Oweight* and k. The experimental data in section 4 was collected using these default settings.

The arithmetic coder represents probabilities as a ratio of an integer bin probability to the sum of all integer bin probabilities. Because the arithmetic coder wants to express probability as a ratio of two integers, it uses the $f_i(t)$ probabilities. Therefore, all terms of $f_i(t)$ are multiplied by S before they are rounded to minimize the loss of precision that occurs when converting the real probabilities to integer probabilities. For example, if $S = 2$, the probability 0.5/2 becomes 1/4 instead of rounding to 1/2.

Also, it is important to note that a new bin gets some probability even though it has no history. *Oweight* is a multiplier that gives observed frequencies greater importance than the probability a bin received before it had a history.

Last but not least, the $b(t)/m(t)$ term in the computation of $g_i(t)$ is a scaling factor so that old bins don't get extra probability just because they've been around a long time.

3.2 String Extension (ZLWAS)

As discussed earlier, each time a ZLW coder emits a code representing a string x, it adds a new string to the dictionary. Miller and Wegman [3] call this character extension because the new string is one character longer than string x. String extension [3] adds to the dictionary the concatenation of the last two dictionary entries for which codes were emitted.

$MaxBlockSize$ = the maximum number of coder emissions from time t to $t+1$.

$MaxBinSize$ = the maximum number of codes per bin at any time

D = the dictionary.

A = the alphabet size, and is 256.

$D(t)$ = the number of strings in the dictionary at time period t, where
$D(0) = A$

$D2(i)$ = the size of the dictionary after i coder emissions.

$m(t) = min(MaxBlockSize, b(t-1))$ is the number of coder emissions in time period t.

M = the total number of emissions from the coder.

$n(t)$ = the maximum number of codes per bin at time t, and is given by
$n(t) = min(MaxBinSize, \lceil (D(t)+1)/b(t) \rceil)$

$b(t)$ = the number of bins at time t, and is given by
$b(t) = max(\lceil \sqrt{D(t)} \rceil, \lceil D(t)/MaxBinSize \rceil)$

Figure 1: Dictionary and Bin Information

String extension makes the dictionary grow faster and contain longer strings, so many sources can be represented by fewer dictionary entries. This idea is used in the ZLWAS coder to allow it to adapt quickly to highly repetitive text when long dictionary entries have a good chance of being used. If the source text makes little use of long dictionary entries, however, string extension can reduce rather than enhance compression. It can increase the number of useless dictionary entries to a point that offsets the gains realized by representing the source text with fewer entries.

3.3 Word Based Strategy (ZLWASC)

One problem mentioned with the string extension enhancement and ZLW coders in general is that they both can create many useless dictionary entries. Conditional dictionary entry is a strategy that often reduces the number of useless

τ = the time constant for frequency predictions, and is 0.9 by default.

S = the scaling factor for integer frequencies, and is 16 by default.

$Oweight$ = the weighting factor for observations and is 5.0 by default.

k = the fraction of codes that are considered new and get probability based on potential, and is 0.1 by default.

$o_i(t)$ = the count of emissions from bin i during time block t.

$a_i(t)$ = the o_i vector adjusted for newly-created bins, where

$$a_i(t) = \begin{cases} o_i(t-1), & \text{for } i \le k * b(t-1) \\ o_{b(t-1)-(b(t)-i)}(t-1), & \text{for } i \ge b(t) - ((1-k) * b(t-1)) \\ o_{b(t-1)/2}(t-1), & \text{otherwise} \end{cases}$$

The scaled integer relative frequency predicted for bin i during time period t is

$$f_i(t) = \begin{cases} round(\tau * f_i(t-1) + (1-\tau) * g_i(t)), & \text{for } t > 1 \\ S, & \text{for } t = 1 \end{cases}$$

and the instantaneous prediction is

$$g_i(t) = \begin{cases} S, & \text{for } a_i(t) = 0 \\ S + round(Oweight * S * a_i(t) * b(t)/m(t)), & \text{otherwise} \end{cases}$$

The sum of relative frequencies at time t is

$F(t) = \sum_{i=1}^{b(t)} f_i(t)$

Figure 2: Bin Probability Information

dictionary entries for a ZLW coder with the string extension enhancement.

The idea behind conditional dictionary entry is that when the entries in a dictionary are letters or parts of words, it makes sense to character extend them until they become words. However, once you have a dictionary of words, it makes little sense to build long phrases character by character. It makes more sense to build them up by putting words together.

The conditional dictionary entry strategy does not add new dictionary entries that end with a blank space when doing character extension. In simple terms, once the dictionary contains a complete word, that word cannot be lengthened by character extension. So the regular ZLWA coder builds up words character by character, while string extension builds up phrases word by word.

If the source contains long runs of blanks, the string extension enhancement will create plenty of long blank strings to make them compress well.

The virtue of this strategy is that it uses a nearly universal word separator, the blank space, as a word boundary. This symbol is uncommon enough in binary files that it does not degrade their compression, and is common enough in program source and text files to enhance compression.

Figure 3 shows the coder with all the outlined enhancements in place.

```
for i := 1 to A do begin
    D_i := chr(i);
end;
IPos := 1;
NumDef := A;
LastString := "";
while IPos <= length(I) do begin
    CodeNum := match(I, IPos, D);
    String := D_CodeNum;
    output(CodeNum, log(f_CodeNum(t)/F(t)));
    IPos := IPos + length(String);
    LastChar := I_IPos;
    if (stringcat(LastString, String) ∋ D) then begin
        NumDef := NumDef + 1;
        D_NumDef := stringcat(LastString, String);
    end;
    if (stringcat(String, LastChar) ∋ D) and (LastChar ≠ SpaceChar) then begin
        NumDef := NumDef + 1;
        D_NumDef := stringcat(String, LastChar);
    end;
    if (m(t) = Σ_{i=1}^{b(t)} a_i(t)) then begin
        t := t + 1;
        UpdateFreqs(f);
    end;
    LastString := String;
end;
```

Figure 3: ZLWASPC Encode Algorithm

3.4 Implementation Decisions

The first coder design decision was choosing the dictionary storage method. Consulting the dictionary must be quick, adding both character and string ex-

tended entries must be quick, and the dictionary should contain as little redundant information as possible.

For dictionary consulting and character extension, a hash trie would provide the best storage method.

This arrangement is inefficient for string extension, however. When doing string extension, both code words are already in the table, and it would be desirable to have an easy way to link them. A system of dual pointers in a tree [3] to signify doubled code words could be implemented, but it was decided that code table consulting speed and implementation simplicity outweighed added storage efficiency and string extension speed. Hence we used a hash trie as in Welch's implementation [5].

The next implementation decision was whether to store frequency data in a sequential array or in a tree. The tree structure offers update or reference of a single entry in $O(\log b(t))$ time, and the array offers update of any number of entries in $O(b(t))$ time, and reference of a single entry in $O(1)$ time.

For a given time period t, the array structure takes

$$m(t)(b(t)/m(t-1)+1)$$

total steps for any time period t. For the same time period, the tree structure takes

$$m(t)\log(b(t))((1+m(t-1))/m(t-1))$$

total steps. When $b(t-1) \leq MaxBlockSize$, $m(t-1) \cong b(t)$, so the array structure takes approximately $m(t)$ steps. In the same situation, the tree structure takes about $m(t)\log(b(t))$ steps. So when $b(t-1) \leq MaxBlockSize$, the array storage method is always better than the tree storage method. When $b(t-1) > MaxBlockSize$, the $b(t)$ term dominates. The array version then takes about $b(t) + MaxBlockSize$ steps, and the tree version takes about $MaxBlockSize\log(b(t))$ steps. So when $b(t-1) > MaxBlockSize$ and $b(t) + MaxBlockSize > MaxBlockSize\log(b(t))$ the tree version is the best storage method.

Experiments indicate that a large bin size is desirable. On even the largest test text files, $b(t-1)$ never came close to $MaxBlockSize$, so for our purposes a sequential array is the best storage choice. Had a small bin size been chosen, the tree structure may have been a better choice.

4 Experiments Using the ZLWAx Coders

Experiments were conducted on a variety of text files:

1. A file of student essays (file *essays*).
2. A Pascal source code file (file *Pascal*).

3. The TEX source to the complete TEX manual (file *TEX*).

4. A file of USENET news articles (file *News*).

5. A file of three repeated characters (file *abc*).

6. An executable program (file *csh*).

Each of these files was compressed using several methods.

1. UNIX compress, a standard implementation of the Lempel-Ziv compression algorithm that encodes dictionary entires using Huffman coding. Two versions of compress are listed: one using a twelve-bit code table (ZL12), and the other a sixteen-bit code table (ZL16). The more bits in the code table, the more codes it can hold.

2. Lempel-Ziv coding where dictionary entries are arithmetically coded (ZLWA).

3. Lempel-Ziv with arithmetic coding and string extension (ZLWAS).

4. Lempel-Ziv with arithmetic coding, string extension and the predictive strategy described above (ZLWASP).

5. Lempel-Ziv with arithmetic coding, string extension, predictive strategy and conditional dictionary entry (ZLWASPC).

The compression results are given in table 3.

File	Size (bytes)	Size and Percent Reduction by Compression Method					
		ZL12	ZL16	ZLWA	ZLWAS	ZLWASP	ZLWASPC
essays	176369	81808 (54%)	70833 (60%)	68298 (61%)	65534 (63%)	64627 (63%)	59725 (66%)
Pascal	119779	43791 (63%)	35756 (70%)	34372 (71%)	20605 (83%)	20089 (84%)	19171 (84%)
TEX	201746	96920 (52%)	79603 (61%)	76718 (62%)	68011 (66%)	66422 (67%)	62567 (69%)
News	210931	141921 (33%)	113665 (46%)	110081 (48%)	102810 (51%)	97242 (54%)	94326 (55%)
abc	150000	1179 (99.2%)	1179 (99.2%)	1119 (99.2%)	46 (99.9%)	40 (99.9%)	40 (99.9%)
csh	122880	91070 (26%)	78697 (36%)	75835 (38%)	72777 (41%)	67164 (45%)	67092 (45%)

Table 3: Compression Results

One point of interest in table 3 is the comparison between the ZLWA and ZLWAS coders. The *Pascal* file in particular shows a large compression gain when string extension is added. Examining the *Pascal* file showed that it started with many similar forward declarations. This high repetition resulted in a substantial compression gain, as expected. The *TEX* file contains many consecutive similar index entries, also giving a healthy compression boost. Repetition also accounts

for the tremendous compression gain realized by adding string extension when compressing file *abc*.

The predictive model (ZLWASP) seemed to work well on the *News* file. This file contains many context shifts due to having articles on varied topics, but also contains periodic repetition because of the header information appearing on every article. It seems that the predictive model gave emphasis to new strings, but also didn't forget that certain strings were liable to be used repeatedly.

Both the *essays* and *TEX* files show the power of conditional dictionary entry in ZLWASPC. Each of these files gained significant compression over ZLWASP by using this simple strategy.

Comparing our algorithms to 16-bit UNIX compress, we find that the ZLWASPC compressed English text and source code files are smaller. The compressed file *essays* is reduced by 16 percent, file *Pascal* is reduced by 46 percent, file *TEX* is reduced by 21 percent and file *News* is reduced by 17 percent. For these four text files, the average improvment for the ZLWASPC coder over 16-bit UNIX compress is 25 percent. When we examine the files containing text that is neither English text nor source code, we still show significant improvment over 16-bit UNIX compress. The compressed file *abc* is reduced by 97 percent, and file *csh* is reduced by 15 percent.

It is also interesting to observe how the string extension and conditional dictionary entry strategies affect the dictionary. Table 4 shows the number of coder emissions M for a coder with $D(T)$ dictionary entries, both with and without the conditional entry and string extension enhancements.

	ZLW, ZLWA		ZLWAS, ZLWASP		ZLWASPC	
File	$D(T)$	M	$D(T)$	M	$D(T)$	M
essays	39624	39368	70897	35732	49584	34661
Pascal	21373	21117	24423	12534	19837	12231
TEX	44009	43753	72496	37041	55191	35908
News	61040	60784	103672	54273	86523	53845
abc	1205	949	340	44	340	44
csh	43556	43300	69946	39858	68310	39902

Table 4: Dictionary Statistics

To interpret the data from table 4 we must consider how dictionary size and number of coder emissions affects the size of a ZLWA coder output file. In cases where no string extension occurs, the dictionary is augmented every time the coder emits the representation of a dictionary entry, so at all times $D2(i) = i + A$. Therefore the output length $L(ZLWA)$ of the ZLWA coder is

given by

$$L(ZLWA) = \sum_{i=A}^{M+A} \log(D2(i)) = O(D2(M)\log D2(M)).$$

In the ZLWAS coders, emissions won't occur as often as the dictionary is added to. On average, emissions will occur with $M/D2(M)$ of dictionary enterings. Thus

$$L(ZLWAS) = \frac{M}{D2(M)} L(ZLWA) = O(M\log(D2(M))).$$

Therefore, to improve compression efficiency we must either reduce the number of coder emissions or greatly reduce the number of dictionary entries.

It is clear from table 4 that string extension, as would be expected, greatly increases the number of code words. Fortunately, in all but one of our test files it reduces the number of coder emissions by enough so that compression is improved.

Adding the conditional dictionary entry enhancement makes string extension an even more powerful strategy for many text files. Table 4 shows that for most of the test files conditional dictionary both greatly reduces the number of dictionary entries and reduces the number of emissions.

Experiments also show that this coder runs reasonably quickly, compressing about 1800 bytes per second on a Sun-3. This speed could be greatly improved by using an optimized arithmetic coder, removing debugging code and removing the statistics collection code. In comparison, the well optimized ZL16 runs at about 27,000 bytes per second on a Sun-3.

5 Conclusions and Directions for Future Work

Each of the strategies described in this paper squeeze significant additional compression out of ZLW coding. These techniques operate quickly enough and can be implemented simply enough to apply them to compression utilities that could be used daily.

Future directions for ZLW-based coders are many, but it would seem that improving the speed of the arithmetic coder, enhancing the conditional dictionary entry strategy or adding a first order Markov model to the predictive strategy are the most promising directions. The current conditional dictionary entry strategy has the virtues of simplicity and applicability to many different kinds of text files, but it could be enhanced to recognize attributes of text files and dynamically adjust conditional entry to improve compression. The existing predictive strategy seems incompatible with the first order Markov conditioning scheme from Miller and Wegman's $A2$ algorithm, but it would be interesting to

modify the predictive strategy to benefit from Markov modeling. Another possibility would be to incorporate Storer and Szymanski's linear time *OMP/UD* scheme [4] into our dictionary entry heuristics.

It would also be interesting to directly compare the ZLWAx coders with powerful coders like Miller and Wegman's *A2* or Cleary and Witten's Markov modeling with partial string matching [1].

Acknowledgments

We gratefully acknowledge Ian Witten, for providing us with online C source for his arithmetic coder. We also benefitted greatly from discussions with James Storer and Mark Wegman.

References

[1] John G. Cleary and Ian H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, COM-32(4):396–402, April 1984.

[2] Debra A. Lelewer and Daniel S. Hirschberg. Data compression. *ACM Computing Surveys*, 19(3):261–296, September 1987.

[3] Victor S. Miller and Mark N. Wegman. *Variations on a Theme by Ziv and Lempel.* Technical Report RC 10630 (47798), IBM, Thomas J. Watson Research Center, July 1984.

[4] James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *Journal of the ACM*, 29(4):928–951, October 1982.

[5] Terry A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, June 1984.

[6] Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, June 1987.

[7] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, IT-24(5):530–536, September 1978.

[8] Jacob Ziv and Abraham Lempel. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, IT-22(1):75–81, January 1976.

[9] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3):337–343, May 1977.