Available online at www.sciencedirect.com

ScienceDirect

Information and Computation 205 (2007) 1012–1026

Information and Computation

www.elsevier.com/locate/ic

Fast gapped variants for Lempel–Ziv–Welch compression ☆

Alberto Apostolico *

Dipartimento di Ingegneria dell' Informazione, Università di Padova, Via Gradenigo 6a, 35131 Padova, Italy
College of Computing, Georgia Institute of Technology, 801 Atlantic Drive, Atlanta, GA 30318, USA

Received 6 May 2005; revised 3 August 2006
Available online 12 March 2007

Abstract

Variants of classical data compression paradigms by Ziv, Lempel, and Welch are proposed in which the phrases used in compression are selected among suitably chosen strings of intermittently solid and wild characters produced by the autocorrelation of the sourcestring. Adaptations and extensions of the classical ZL78 paradigm as implemented by Welch are developed along these lines, and they are easily seen to be susceptible of simple linear time implementation. Both lossy and lossless schemata are considered, and preliminary analyses of performance are attempted.

Keywords: Data compression by textual substitution; Lossy compression; Ziv–Lempel–Welch phrase parse; Pattern matching; Trie; Design and analysis of algorithms

1. Introduction

Ziv and Lempel designed a class of compression methods based on the idea of self reference: while the textfile is scanned, substrings or *phrases* are identified and stored in a *dictionary*, and whenever, later in the process, a phrase or concatenation of phrases is encountered again, this is compactly encoded by suitable pointers or indices [11,20,21]. Of the existing versions of the method, we recapture below Welch's implementation of [21], which we shall refer to as *LZW*, incarnated in the `compress` command of UNIX. This version is reported in Fig. 1.

For the encoding, the dictionary is initialized with all the characters of the alphabet. At the generic iteration, we have just read a segment s of the portion of the text still to be encoded. With σ the symbol following this occurrence of s, we now proceed as follows: If $s\sigma$ is in the dictionary we read the next symbol, and repeat with segment $s\sigma$ instead of s. If, on the other hand, $s\sigma$ is not in the dictionary, then we append the dictionary index

☆ Work supported in part by the Italian Ministry of University and Research under the National Project FIRB RBNE01KNFP and PRIN "Combinatorial and Algorithmic Methods for Pattern Discovery in Biosequences", by the Bi-National Project FIRB RBIN04BYZ7, and by the Research Program of the University of Padova. An extended abstract related to this work appears in the *Proceedings of the IEEE Data Compression Conference, DCC 2005*, Snowbird, Utah, March 2005.

* Fax: +1 39 49 828 76.
E-mail address: axa@dei.unipd.it

0890-5401/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.ic.2007.03.001

Initializations
Initialize dictionary trie the single characters from Σ
Initialize phrase s to first input character
Set $output \leftarrow \langle code(s) \rangle$
Body Repeat until no more input characters
$\sigma \leftarrow$ read the next input character
if $s\sigma$ is in the dictionary then set $s = s\sigma$ (*seek-phrase* continues)
else (the end of a stored phrase has been reached)
1- $output \leftarrow output \cdot \langle code(s) \rangle$
add $s\sigma$ to dictionary
2- $s \leftarrow \sigma$
end Body

Fig. 1. LZW.

⟨0⟩	⟨1⟩	⟨3⟩	⟨0⟩	⟨0⟩	⟨2⟩	⟨7⟩	⟨1⟩	⟨8⟩	⟨3⟩	⟨2⟩	⟨1⟩	⟨14⟩	⟨7⟩
a	b	ab	a	a	c	ac	b	ca	ab	c	b	bb	ac
ab	ba	aba	aa	ac	ca	acb	bc	caa	abc	cb	bb	bba	aca
3	4	5	6	7	8	9	10	11	12	13	14	15	16

Fig. 2. **Top Half:** Illustrating LZW parse and encoding as applied to the string *abababaacacbcaabcbbbac*. Each integer in the sequence of the top row is the encoding of the phrase appearing immediately beneath it, in the second row. The third row shows the new entry that is added to the trie once the phrase is parsed that appears immediately above that entry. The last row shows the integer encoding of the trie entry immeditaely above it. **Bottom Half:** Lossless LZWA parse for the same string. Here, a pair (i, j) in the first row indicates the encoding of a phrase as resulting from the shuffle of the ith phrase of the dictionary and the jth resolver ($j = 0$ means no resolver).

of s to the output file, and add $s\sigma$ to the dictionary; then reset s to σ and resume processing from the text symbol following σ. Once s is initialized to be the first symbol of the source text, "s belongs to the dictionary" is established as an invariant in the above loop. As an example, the string $s = abababaacacbcaabcbbbac$ is parsed as in the top section of Fig. 2. Note that the resulting set of phrases or codewords obeys the *prefix closure* property, in the sense that if a codeword is in the set, then so is also every one of its prefixes. The figures illustrate this operation.

LZW is easily implemented in linear time using a trie data structure as the substrate [20,21], and it requires space linear in the number of phrases at the outset. Another remarkable property of LZW is that the encoding and decoding are perfectly symmetrical, in particular, the dictionary is recovered while the decompression process runs (except for a special case that is easily taken care of).

Lossy compression by textual substitution has been variously defined and studied in recent years. We refer to, e.g., [7,12] for recent surveys of motivation and results. Most of the lossy extensions of ZL schemata proceed by finding a best match, within an assigned fidelity, between the incoming stream and the already seen portion of the text or some suitably pre-arranged external dictionary. Irrespective of the criterion used to find such an optimum match, the search itself is inherently time consuming. From the standpoint of computation, the reason for this resides ultimately with the fact, that pairwise matching of symbols is transitive while mis-matching is not. One more difficulty may be introduced by the need to preserve the information relative to the positions of mismatches in a phrase, which poses extra overhead in encoding. The state of the art and challenges ahead are nicely captured in [7]:

> "All universal lossy coding schemes found to date lack the relative simplicity that imbues Lempel–Ziv codes and arithmetic codes with economic viability. Perhaps as a consequence of the fact that approximate matches abound whereas exact matches are unique, it is inherently much faster to look

> for an exact match that it is to search from a plethora of approximate matches looking for the best, or even nearly the best, among them. The right way to trade off search effort in a poorly understood environment against the degree to which the product of the search possesses desired criteria has long been a human enigma. This suggests it is unlikely that the "holy grail" of implementable universal lossy source coding will be discovered soon."

The emphasis in this paper is on speed of encoding and decoding, which will rest in particular on an effective, implicitly encodable choice of the gap positions in a phrase. The family of methods considered is inspired by the notion of a *motif*. In a motif driven LZW parse, dictionary entries correspond to strings written on the alphabet $\Sigma \cup \{\_\}$, where "_" represents a wildcard or "don't care" symbol that may take up one of several specifications. Allowing don't cares leads to an increase of the average length of phrases hence to a decrease in their number. As the sample tables reported here from [5,2,3] display, savings can be dramatic for inputs such as images and signals, in which the gaps can be filled by interpolation at the receiver with negligible loss. As it turns out, however, improvements are also seen in lossless schemata in which a dictionary of *resolvers* must be added to disambiguate the don't cares in the phrases.

We are also interested in assessing performance, expressed in terms of asymptotic compression bounds and related coding theorems, though this seems less easy in our framework. To enucleate a rough rule of thumb, one might compare encodings under the rosiest possible scenario, that is, by assuming that allowing k don't care in each phrase will increase by k the size of each phrase and hence reduce by a factor of k the total number of phrases. In other words, for every iteration of the standard LZW parse, the parser would now keep going k times and skip as many commas. In lossless implementation, however, this will come at a cost of doubling the code for each phrase. In fact, as will be seen in Section 3, in this case we need to transmit, together with the code of each phrase (representing a string on $\Sigma \cup \{\_\}$), also the code of the resolver string (on Σ) needed by the decoder to fill the gaps. As is known (see, e.g., [11]) , if t is the number of phrases produced by the standard LZW for an input of size n, the number L of bits in the corresponding encoding is

$$L \approx (t)\log(t)$$

Under our assumptions, the output size would become

$$L' \approx 2\left(\frac{t}{k}\right)\log\left(\frac{t}{k}\right)$$

thereby inducing a *gain* expressed, e.g., as

$$\frac{L'}{L} \approx \frac{2}{k}\left(1 - \frac{\log(k)}{\log(t)}\right)$$

which is positive for $t > k$. The last expression gives qualitative insight into the gain that can be expected. In practice, things are different. To begin with, if a phrase is now the concatenation of $k+1$ shorter phrases, the chance of this extended phrase appearing again in the text is decreased.

Both in lossless and lossy implementations, having to specify where the don't cares reside within each phrase would be simply prohibitive. The phrases used in the present paper are a deterministically generated set of approximate patterns with don't cares, that result from the self-correlation of the source-string. In so far as this rigidly dictates the structure of phrases, it might affect the compactness of the encoding; in exchange, it affords computation in linear time. The particular choice made is inspired by recently developed combinatorial notions and properties that set the framework for picking don't cares. These are highlighted next. We refer to [4] for a more formal treatment.

A *motif* w in a string x is an element of Σ or any string on $\Sigma \cdot (\Sigma \cup \{\_\})^* \cdot \Sigma$ *together* with its list of occurrences $\mathcal{L}_w$ in x. A motif that occurs at least q times in x is a *q-motif*. Motif w is *maximal* if we cannot make it more specific or longer by adding solid characters while retaining $\mathcal{L}_w$ (up to a uniform offset $d, |d| \geq 0$). A maximal motif w is *irredundant* if w and $\mathcal{L}_w$ cannot be deduced by the union of a number of lists of other maximal motifs. Let now x and y be two strings with $m = |x| \leq |y| = n$. The *consensus* of x and y is the string $z_1 z_2 \ldots z_m$ on $\Sigma \cup \{\_\}$

defined as: $z_i = x_i$ if $x_i = y_i$ and $z_i =$"_" otherwise ($i = 1, 2, ..., m$). Deleting all leading and trailing don't care symbols from z yields a (possibly empty) *2-motif* that is called the *meet* of x and y. A set $\mathcal{B}$ of motifs capable of generating all other 2-motifs in a string is called a *basis*. A linear-sized *basis* of irredundant 2-motifs can be built incrementally in time $O(n^3)$ using the property that every irredundant 2-motif in s is the meet of two suffixes of s [4]. Motif-based *off-line* algorithms were introduced and tested in [2,3,5]. The majority of those applications assume a preprocessing to extract motifs from the textstring.

The remainder of the paper is organized as follows. In the next section, we evoke a match length and recurrence time backstage for our algorithms. Next, we develop and analyze the algorithm of Fig. 3 and some of its variants. Following that, we compute projected compressions in correspondence with extremal sequences. Some tables reporting from experiments are given after that, and a list of conclusions and plans for future work close the paper.

2. Match lengths

Most of the lossy extensions of ZL schemata proceed by finding a best match, within a given fidelity, between the incoming stream and the already seen portion of the text or some suitably pre-assigned external dictionary. Possible schemes along these lines include, e.g., adaptations of those in [14], or more radical schemes in which the innovative add-on inherent to LZW phrase growth is represented not by one symbol alone, but rather by that symbol plus the longest match with the substring that follows some previous occurrence of the phrase. We go back to the rough estimate made at the beginning and consider the expected length of a phrase assuming a maximum density d of mismatches per phrase. More formally, we will take the per-symbol *distortion* $d(w, \hat{w})$ for two strings of m characters to be the average Hamming Distance

$$d = \frac{1}{m} \sum_{i=1}^{m} d(w_i, \hat{w}_i)$$

where $d(w_i, \hat{w}_i) = 1$ if $w_i \neq \hat{w}_i$ and $d(w_i, \hat{w}_i) = 0$ if $w_i = \hat{w}_i$. An average *fidelity* $f = 1 - d$ is naturally associated with a distortion of d.

For fixed d or f assuming a source that emits symbols with an iid distribution, consider two strings of m characters and set

$$p = \sum_{i=1}^{|\Sigma|} p_i^2$$

the average probability of a single match, where p_i is the probability associated with the ith character of the alphabet. Then, the probability of a match at distortion d or fidelity f between the two strings is

$$B(m, p, f) = p^{(mf)}(1-p)^{(m-mf)} \binom{m}{mf} = p^{(mf)}(1-p)^{(md)} \frac{m!}{(md)!(mf)!}.$$

Using Stirling approximation $\ln(n!) = n(\ln n - 1)$ on the logarithm of $B(m, p, f)$, and going back by exponentiation we get, after easy passages:

$$B(m, p, f) = \mathrm{e}^{-m\mathcal{H}(f,p)}$$

where $\mathcal{H}(f, p)$, defined as

$$\mathcal{H}(f, p) = f \log\left(\frac{f}{p}\right) + (1-f) \log \frac{(1-f)}{(1-p)} + c$$

with c a constant, is the *relative entropy* or Kullback–Liebler distance between the f and p "coins". $\mathcal{H}(f,p)$ is always positive, as a consequence of Gibbs theorem, well known to physicists, according to which for two distributions (actually, sets of real numbers of same sum) p_i and q_i

$$-\sum_{i=1}^{n} p_i \log p_i \leq -\sum_{i=1}^{n} p_i \log q_i.$$

We may estimate the length of a longest match at distortion d between the incoming phrase and those already in the dictionary by computing the longest such match that occurs anywhere between two independent, left justified random sequences. Formally, for two random sequences $X_1, X_2, ..., X_i, ...$ and $Y_1, Y_2, ..., Y_i, ...$, let $C_i \equiv \mathcal{I}(X_i = Y_i)$ be defined to have value 1 if $X_i = Y_i$, $(i = 1, 2, ..., j, ...)$ and 0 otherwise, and let

$$R_n^f = max\left\{ m : fm \leq \sum_{1\leq k\leq m} C_{i+k}, 0 \leq i \leq n-m \right\}.$$

By a known result by Erdös–Rényi, when the C_i's are independent Bernoulli variables with probability p and $p < f \leq 1$, we have that

$$P\left(\frac{R_n^f}{\log(n)} \to \frac{1}{\mathcal{H}(f,p)}\right) = 1.$$

In intuitive terms, this can be derived by taking $e^{-m\mathcal{H}(f,p)}$ as the probability of a dense headrun of length m in a series of coin tossing, and forcing such a headrun to occur with certainty at any one of about n starting positions. Thus, setting:

$$1 = ne^{-m\mathcal{H}(f,p)}$$

we obtain

$$R_n^f = m = \frac{\log(n)}{\mathcal{H}(f,p)}.$$

Note the analogy of the above with the Asymptotic Equipartition Property for an i.i.d. random source, which states that the logarithm of the inverse of the probability (which is, by Kac's Lemma, the expected logarithm of the recurrence time) of a string, divided by its length, is close to the entropy [8,17].

It is easy to see that forfeiting the search for the *longest* match has the undesired effect of bringing the size of the expected match length down to a constant. With

$$p = \sum_{i=1}^{|\Sigma|} p_i^2$$

the probability of a single match, the probability of a match of length exactly m is $p^m(1-p)$. The corresponding average value or expected length is $\sum mp^m(1-p)$ and is computed according to the formula

$$\sum kx^k = \frac{x}{(1-x)^2}$$

for $|x| < 1$. Hence, the expected length is the well known $p/(1-p)$.

Allowing a density of d for don't cares, the probability of one match becomes $[d + (1-d)p]$, whence that of exactly m matches becomes

$$\hat{P} = [d + (1-d)p]^m(1-d)(1-p) = [1-(1-p)(1-d)]^m(1-d)(1-p)$$

We can write

$$\hat{P} = (1-d)(1-p)\sum_{k=0}^{m}\binom{m}{k}[(p-1)(1-d)]^k.$$

Note that since $x = d + (1-d)p$ is a probability (the probability of a single match) we must have

$$0 \le x \le 1$$

hence

$$-1 \le x - 1 = (p-1)(1-d) = (d-1)(1-p) \le 0.$$

If we stipulate that $|x| < 1$, then we can use the fact that for any (integer or non-integer) m

$$\hat{P} = (1-x)(1+x)^m = (1-x)\sum_{k=0}^{m}\binom{m}{k}x^k = \sum_{k=0}^{\infty}\binom{m}{k}x^k.$$

If now, in

$$\sum kx^k = \frac{x}{(1-x)^2}$$

we set $x = d + (1-d)p$ we get the new expected match length

$$\hat{m} = \frac{d + (1-d)p}{(1-p)(1-d)}.$$

Thus the ratio of the average match allowing a distortion of d over that of no distortion is

$$\frac{\hat{m}}{m} = \frac{d + (1-d)p}{(1-p)(1-d)} \cdot \frac{(1-p)}{p} = \frac{d + (1-d)p}{p(1-d)} = 1 + \frac{d}{p(1-d)}.$$

The discussion of this section seems to suggest that finding a *longest* match is crucial to parsing schemata that aimed at asymptotic entropy rate. However, one key factor that presides over the performance of ZL methods is the *distinctness* of phrases in a parse [8,11]. In the next section, we introduce a mechanism, patterned after LZW, for producing in linear time both lossy and lossless parses consisting of distinct phrases which deterministically allocate gaps and the distortion that goes with them.

3. Motif driven ZL compression

As mentioned, motif based off-line compression show good performance [2,3,5] on a variety of inputs. Those approaches may be regarded as "external dictionary" schemes, in that we start with a set of irredundant motifs already given. It is natural to inquire into the structure of ZL and LZW parses which would use these patterns in lieu of exact strings. In other words, the task of vocabulary build-up is assigned to the growth of (candidate), perhaps irredundant, 2-motifs. We will now first examine briefly the "natural" adaptations along these lines of the two main paradigms of Ziv–Lempel compression, respectively, known as ZL77 and ZL78 [20,21] (for the second one, we consider its subsequent implementation by Welch [16] desribed at the beginning). Both variants could have lossy and lossless versions, which should be dealt with separately. Once the implications of those schemes will be understood, we shall be ready for a more radical approach.

At the generic step, ZL77 has parsed and encoded the first $i-1$ symbols of the source x and looks for the longest prefix of the remaining suffix that matches some previously seen substring. With j the starting position of that substring, l its length and σ the character immediately following it, the new phrase in the parse is encoded

by the triplet (j, l, σ). In our lossy variant, we must assume that a maximum acceptable density d of don't cares, is prescribed. At the generic step, we thus look for the densest 2-motif beginning at i and at some $j < i$ and encode it by a triplet in much the same way as in the traditional ZL scheme. For the lossless variant, the threshold d might or might not be defined, but we must encode, in addition to the triplet, the mismatching characters *along with their positions*. It may be expected that this overhead will take too high a toll on the encoding, whence this scheme might be viable for lossy compression only. There, don't cares can be left unresolved or copied from an interpolated or erroneous character.

We look next at possible adaptations of LZW, Welch's implementation of ZL78. As seen, dictionary entries correspond here to strings written on the alphabet $\Sigma \cup \{\_\}$ (the dictionary is still initialized only by characters of Σ, as before). To find the next phrase in the parse, beginning, say, at position i of the text, we look for *a* longest phrase in the dictionary that matches the still un-parsed suffix of the source and *appearing at least twice so far* (to trigger this invariant condition, it suffices to prepend the string of symbols of Σ to the source string x). Let s be this phrase and h its index in the dictionary. We use the reference number h to encode the current phrase. The next step is the augmentation of the dictionary of phrases. For this, we look for a longest match with up to k mismatches (or within a maximum density of d) between the suffix starting at i and the one that begins at the leftmost occurrence of w. (If this string extends beyond i then it is truncated at $i - 1$, to secure self-feeding in decoding.) We may create an extension in the dictionary corresponding to this new word, and give it a suitable index. In the lossless implementation of this scheme, we would also have to append to this node a list of positioned characters that distinguish, and will enable to reconstruct, the new occurrence from the old one. Like for the ZL77 scheme, this might be bulky in general. To mitigate this problem, a new phrase could be described by the pair of indices of two past phrases. Still, the encoding of a single phrase doubles in format and probably in size in this way. As seen next, a more compact encoding stems from a closer adherence to the original parse.

The generic stage of LZW may be considered as consisting of two parts, as follows. In the first part (hereafter, *seek phrase*), a longest matching phrase from the dictionary is found, and its index is appended to the output. In the second, the one-symbol extension of the *current occurrence* of the phrase is added to the dictionary for possible future reference. In our lossless adaptation, Part 1 is identical, except that the output now must contain both the reference to the phrase and also the characters for its disambiguation. This is achieved through the use of two dictionaries that grow hand-in-hand, whereby a phrase is resolved in the shuffle of a word from the main dictionary and one from the auxiliary one. Looking for a *best* phrase, e.g., the one minimizing mismatches is feasible but time consuming, hence our implementation of seek-phrase greedily pursues matches over mismatches. If we assume $\Sigma \cup \{\_\}$ to be sorted with "_" its maximum element, then seek-phrase finds the lexicographically least phrase occurring at the current position.

The pseudo-code of Fig. 3 describes the algorithm, called *LZWA*. At the outset, each phrase in the parse is decomposed in a pair consisting of a primary phrase s over $\Sigma \cup \{\_\}$ and an auxiliary resolver s' over Σ. A suitable shuffle of s and s' reconstructs the actual phrase over Σ. The information needed for the shuffle is the set of positions of s occupied by don't cares. However, this does not need to be supplied explicitly. Rather, the gaps are filled with the characters from s', in exact succession, and the mechanics of the encoding and decoding, respectively, takes care of consistency.

For an example, Fig. 4 traces the parse and encoding of a string beginning by *ababaacabab*.... The first phrase, *a*, is found in the initial trie at node labeled 0. The trie is expanded by the path *ab*, which receives a code of 3 (since there are already nodes 1 and 2 corresponding to *b* and *c*), and the output pair $\langle 0, \lambda \rangle$ is emitted, where λ denotes that there is no resolver. The second phrase similarly follows the path to node *b*, which is labeled 1, expands this path to *ba*, labeled 4, and outputs the pair $\langle 1, \lambda \rangle$. The next phrase is *ab*, at node 3, which is expanded into path *aba* to a new node 5. The pair $\langle 3, \lambda \rangle$ is the output. When the suffix *aac*... is read into the trie, this creates a don't care transition from Node 0 to a new node labeled 6, since node 5 already had a child but on an arc labeled by *b*. The corresponding output is $\langle 0, \lambda \rangle$. The next phrase is *ac*. This phrase reaches node 6, using the path *a_* and the resolver *c* which exists in the resolver trie by initialization. Node 6 is expanded into a new node 7, on the path *a_a*, and the pair $\langle 6, 2 \rangle$ representing the node 6 of the primary trie and node 2 in the resolver trie is output. The next phrase is *aba*, which can use the path *a_a* to node 7, creates the new node 8 for the path *a_ab* and outputs $\langle 7, 1 \rangle$. The reader is encouraged to continue the example to a point where also the resolver trie starts to be expanded.

Initializations
Initialize dictionary trie and resolver trie with the characters from Σ
Initialize phrase s to first input character, resolver s' to the empty word λ
Set $output \leftarrow \langle code(s), code(s')\rangle$
Body Repeat until no more input characters
$\sigma \leftarrow$ read the next input character
if $s\sigma$ is in the dictionary then set $s = s\sigma$ ($seek\text{-}phrase$ continues)
else, if s_- is in the dictionary and $s'\sigma$ is in resolvers
then set $s = s_-$, $s' = s'\sigma$ ($seek\text{-}phrase$ continues)
else (the end of a stored phrase has been reached)
1- $output \leftarrow output \cdot \langle code(s), code(s')\rangle$
2- if $s\sigma'$ for some $\sigma' \neq \sigma$ is in the dictionary then
2a- if s_- is not already in dictionary add s_- to it
2b- if $s'\sigma$ is not already in resolvers add $s'\sigma$ to it
else add $s\sigma$ to dictionary
3- $s \leftarrow \sigma$
end Body

Fig. 3. LZWA: a motif-driven LZW.

$\langle 0,\lambda\rangle$	$\langle 1,\lambda\rangle$	$\langle 3,\lambda\rangle$	$\langle 0,\lambda\rangle$	$\langle 6,2\rangle$	$\langle 7,1\rangle$	...
(1,0)	(10,3)	(6,2)	(12,3)	(14,2)		
a	b	ab	a	ac	aba	b...
ab	ba	aba	a-	a-a	a-ab	...
3	4	5	6	7	8	9..
				2(c)	1(b)	...

Fig. 4. **Top Half:** Illustrating LZW parse and encoding as applied to the string *ababaacacbcaabcbbbacabbb*. Each integer in the sequence of the top row is the encoding of the phrase appearing immediately beneath it, in the second row. The third row shows the new entry that is added to the trie once the phrase is parsed that appears immediately above that entry. The last row shows the integer encoding of the trie entry above it. **Bottom Half:** Lossless LZWA parse for a string *ababaacabab*... Here, a pair $\langle i, j\rangle$ in the first row indicates the encoding of a phrase as resulting from the shuffle of the ith phrase of the dictionary and the jth resolver ($j = \lambda$ means no resolver). The second row tracks the node expansions of the primary trie performed at each pass, the third row the corresponding codes. The last row shows the code of the resolvers used in the encoding of the first row.

As is easy to see, each phrase is a subsequence of a meet although not necessarily a maximal or irredundant motif. There are $O(n^3)$ possible such motifs, but the parsing strategy learns a sublinear subset of this set. The following easy claims systematize two basic facts.

Theorem 1. *Algorithm LZWA works in time and space linear in the source text x.*

Proof. Inherited straightforwardly from LZW. □

Theorem 2. *Except possibly for the root, the dictionary trie built by LZWA is a binary tree, in which each internal node has either one arc labeled by some character from* Σ *or two arcs, one labeled by a character from* Σ *and one by the don't care "_".*

Proof. We discuss the operation and terminating condition for LZWA. Let $\bar{x}$ be the suffix of the source yet to be encoded. The search for the new phrase can be segmented into three main parts. In Part 1, we seek the longest path in the trie that matches a prefix of $\bar{x}$. This part terminates in one of two possible ways, which we call Mode A and B, respectively, depending on whether the last attempted transition following s reached a leaf or an internal node. In the first case, all conditions must fail so that the last assignment under 2 is reached and the dictionary trie is extended with a single new arc labeled by σ, hence by a solid character. In Mode B, s ends at an internal node, call it v, and we cannot have simultaneously that $s\_$ is in the dictionary and $s'\sigma$ is in the resolvers. If one cause for termination is that $s\_$ is not in the dictionary, then inductively v has one child and the corresponding arc must be labeled by some $\sigma' \neq \sigma$: the algorithm adds an arc labeled "_" to v, which makes v a binary node, and predisposes a resolver $s'\sigma$ for the future. On the other hand, if the default is on $s'\sigma$ not being in the resolvers, then inductively v has already two children (one labeled $\sigma' \neq \sigma$ and the other labeled "_"), and the action of the algorithm is limited to adding $s'\sigma$ to the resolvers. □

We leave it as an exercise to show that correct decoding is possible both for the lossless as well as the lossy encoding, and it works in linear time. It is worth to pin-point a few modifications and upgrades of LZWA that have no substantial bearing on time performance. To begin with, the condition that each new phrase must be able to fill its gaps using an entry of the resolver trie may be forfeited in lossy compression: such a condition slows down the process of phrase vocabulary growth, which may constitute an undesirable bottleneck. Other noteworthy variations are as follows.

- **[cheaper encoding of resolvers]** Since the vocabulary of resolvers grows somewhat independently of that of phrases, it may happen occasionally that the current resolver s' is a prefix of another one, say, s'', already existing in the resolver trie. During decoding, the length of s' is inferred from the structure of the phrase s so that a pointer to s'' is enough to retrieve s'. This suggests to dynamically assign distinct encodings only to the leaves of the resolver trie, which is done through an obvious mechanism of inheritance: whenever a node is expanded in a leaf, the leaf inherits the encoding of that node, and when a leaf is branched out of a node, a new code is introduced and assigned to that leaf. With this, if we now call $ext(s')$ any of the longest current extensions of resolver s', then the encoding line may be rewritten as

  1 - $output \leftarrow output \cdot \langle code(s), code(ext(s'))\rangle$.
- **[error density bound]** It is easy to further modify the algorithm in such a way that a maximum number k of errors (third column in Tables 1 and 2) is allowed in each phrase. The sample tables reported here are indicative of results obtainable by this method for lossy schemes in which don't cares are not filled. The reduction in the number of phrases is general and dramatic in some cases. Correspondingly, the length of an entry in the auxiliary table becomes bounded and may be block-encoded, which might appear to be profitable. In the cases considered, however, the toll exacted by heavier phrase encodings tended to offset this gain.
- **[forcing each phrase to be a consensus]** In this version of the algorithm, we make sure that every phrase used in the encoding is a substring of the consensus of two suffixes of the source. For this, we need to carry along in encoding the dictionary of solid phrases which result from the shuffle of phrases and resolvers used so far. Denoting by $s \bowtie s'$ the phrase that corresponds to the shuffle of phrase s and resolver s', we have that seek-phrase is conditioned on

  if $s\_$ is in the dictionary and $s'\sigma$ is in resolvers and $s \bowtie s'$ is in phrases

- **[h-ary dictionary tries with $h > 2$]** In a nutshell, this consists of issuing a don't care transition not the second, but the hth time a same dictionary node is reached. It leads to a bigger dictionary component of the codebook and to a correspondingly smaller resolver trie. This allows one to control the fidelity in lossy variants where resolvers are forfeited, and to determine the best lossless performance by changing the setting of h.

The tables, reported from [2], are indicative of performances obtained in preliminary experiments with some of the above variants. Column 4 shows different values of the maximum allowed number of don't care per phrase,

Table 1
Comparing performances on gray-scale images

Source file	Orig len	LZW phrases	Max d.c.	Lossless phrases	Phrases % diff	Lossy size	Gzip size	Gzip % diff
Camera	66,336	31,575	3	22,145	–29.87%	29,620	48,750	–39.24%
	66,336	31,575	5	21,685	–31.32%	29,616	48,750	–39.25%
	66,336	31,575	20	21,667	–31.38%	29,606	48,750	–39.27%
Bridge	66,336	39,457	3	26,095	–33.86%	34,166	61,657	–44.59%
	66,336	39,457	5	26,094	–33.87%	34,166	61,657	–44.59%
	66,336	39,457	20	26,094	–33.87%	34,166	61,657	–44.59%
Lena	262,944	121,054	20	84,890	–29.87%	117,978	121,054	–49.70%

Table 2
Results with DNA sequences

Source file	Orig len	LZW phrases	Max d.c.	Lossless phrases	Phrases % diff	Lossy size	Gzip size	Gzip % diff
Spor EarlyII	25,008	5,356	4	4,587	–14.36%	4,397	8,008	–45.09%
	25,008	5,356	5	4,192	–21.73%	4,559	8,008	–43.07%
	25,008	5,356	20	4,098	–23.49%	4,673	8,008	–41.65%
Spor EarlyI	31,039	6,446	4	5,662	–12.16%	5,301	9,862	–46.25%
	31,039	6,446	5	5,083	–21.14%	5,471	9,862	–44.52%
	31,039	6,446	20	4,945	–23.29%	5,586	9,862	–43.36%
Spor Middle	54,325	10,409	20	7,887	–24.23%	9,528	16,395	–41.88%
Spor All	222,453	35,920	4	37,759	5.12%	31,477	68,136	–53.80%

the other columns refer to sizes, the number of phrases produced by the original LZW and the new parse, respectively, the corresponding percentage savings in terms of phrases and size. In general, lossless completions do not achieve significant gain due to the overhead of the encoding. By contrast, filling gaps by straightforward interpolation of adjacent pixels is seen to restore images within negligible difference from their corresponding originals.

4. Comparing vocabulary build-ups

In experiments, the tradeoff between LZW and the lossless LZWA alternates, while lossy LZWA brings considerable savings. An analytical comparison of the compression performances associated with LZW and LZWA parses seems not trivial. Under the restriction that the maximum number k of mismatches allowed in each phrase is a predefined constant, it is possible to show [22] that for a string of n characters, as $n \to \infty$,

$$\widehat{t} \geq t\left(1 - 2\frac{k\log^2|\Sigma|}{H\log n}\right)$$

where H is the entropy of the text, t is the number of phrases in ZL78, and $\widehat{t}$ is the number of phrases in a lossy compression of the original sequence which allows up to k don't care symbols per phrase.

In this section, we study the relationship between the maximum number of phrases achievable by LZW and LZWA, respectively, over inputs of equal size. But first note that even the lossy (i.e., with resolvers discarded) version of LZWA does not always save over LZW. For this, consider the extreme case of a string a^n formed by n identical symbols. The number of distinct substrings in such a string is only n. The string achieves a better compression under LZW, and this also shows that the transition to LZWA is not advantageous for all inputs.

In fact, a^n is parsed in almost the same way by LZW and LZWA, roughly producing a number of phrases in the parse equal to

$$n = \sum_{i=1}^{t} i, \quad \text{i.e.,} \quad n \approx t^2/2 \quad \text{whence} \quad t \approx \sqrt{2n}$$

which corresponds to an encoding $t \log t \approx 1/2\sqrt{2n} \log n = 0.71\sqrt{n} \log n$. LZWA does not introduce any extra phrases in the dictionary, and yet it brings about an overhead on the encoding which must now take into account the symbol "_". This costs at least one extra bit.

The situation is different when LZW and LZWA are compared on their respective "most incompressible" input, i.e., on sequences the parsing of which results in the largest number of phrases. We conform to some of the notation and treatment in [11,20,21]. With $|\Sigma| = \alpha$, consider a string formed by the sequence of all words of length 1, 2, 3, ... , k, in lexicographic order. The length of such a string is

$$n_\alpha(k) = \sum_{i=1}^{k} i\alpha^i = \frac{\alpha}{\alpha - 1}\left[\alpha^k\left(k - \frac{1}{\alpha - 1}\right) + \frac{1}{\alpha - 1}\right]$$

which for $\alpha = 2$ becomes, in particular

$$n_2(k) = (k-1)2^{k+1} + 2$$

or, approximately,

$$n_2(k) \approx k2^{k+1}$$

and for α large enough such that $\alpha \approx \alpha - 1$ becomes $n_\alpha(k) \approx k\alpha^k$. For $\alpha = 3$, we have

$$n_3(k) = \frac{3}{2}\left[3^k\left(k - \frac{1}{3}\right) + \frac{1}{2}\right] = \frac{1}{2}\left(k3^{k+1} - 3^k + \frac{3}{2}\right)$$

whence for $k > 0$:

$$\tfrac{1}{2}k3^k < n_3(k) < \tfrac{1}{2}k3^{k+1}.$$

Thus, we can state in general that as α increases from small values to large values $n_\alpha(k)$ goes from $\Theta(k\alpha^{k+1})$ to $\Theta(k\alpha^k)$.

The number of distinct substrings in any string of the type considered is the maximum possible for that alphabet and length. This number is:

$$N_\alpha(k) = \sum_{i=1}^{k} \alpha^i = \frac{\alpha}{\alpha - 1}(\alpha^k - 1)$$

in general, which becomes approximately α^k for large alphabets and $2^{k+1} - 2 \approx 2^{k+1}$ for the binary alphabet. Thus, we can state that in general the vocabulary size $N_\alpha(k)$ of our string goes from $\Theta(\alpha^{k+1})$ to $\Theta(\alpha^k)$ in the transition from a very small alphabet to a very large one.

We may regard the introduction of don't cares as a means of collapsing the alphabet of the main trie, in the sense that once the current phrase is found to diverge from its path in the trie, this creates a don't care transition that will allow any other phrase reaching that junction in the future to expand into a longer match. Consequently, the expectation is that the input string is partitioned into longer phrases and there will be correspondingly fewer of them. In lossless implementations, this gain seems offset by the increased cost of phrase encoding. In lossy schemes, however, we forfeit resolvers and the question becomes natural as to what savings on the number of phrases is brought about by this collapse of the alphabet. This kind of analysis seems not easy in general. However, one rough estimate is offered by the comparison of the values taken by $N(k)$ for comparable

Table 3
Phrase vocabulary growth for $\alpha = 10$ and $\alpha = 2$

k	$n_{10}(k)$	$N_{10}(k)$	$N_2(\bar{k})$	$n_2(k)$	$\bar{k}$	$n_2(\bar{k}+1)$
1	10	10	4	2	1	10
2	210	100	30	98	4	258
3	3210	1,000	254	1538	7	3586
4	43210	10,000	4,094	40962	11	90114
5	543210	100,000	32,766	425986	14	917506
6	6.54321e+006	10^6	262,142	4.19431e+006	17	8.9129e+006
7	7.65432e+007	10^7	2,097,150	3.98459e+007	20	8.38861e+007
8	8.76543e+008	10^8	33,554,430	7.71752e+008	24	1.61061e+009
9	9.87654e+009	10^9	268,435,454	6.97932e+009	27	1.44955e+010
10	1.09877e+011	10^{10}	$2.14748e+009$	6.2277e+010	30	1.28849e+011
15	1.65432e+016	10^{15}	$2.81474e+014$	1.29478e+016	47	2.64586e+016
20	2.20988e+021	10^{20}	$1.84467e+019$	1.1437e+021	63	2.32429e+021
40	4.4321e+041	10^{40}	$2.72225e+39$	3.51171e+041	130	7.07787e+041

input lengths but under different alphabet sizes. As an example, Table 3 summarizes the phrase dictionary sizes (approximated by 10^k and $2^{\bar{k}+1}$, respectively) that correspond to strings of approximately the same length, for decimal and binary alphabets.

Analytical formulas are harder to come by. The approach that follows is based on the observation, that while the trie of phrases produced by LZW on a string over an alphabet of α characters is an α-ry trie, the trie produced by LZWA is always structured as a binary trie even though its arcs take up in general $\alpha + 1$ distinct values. Thus, for a given input length n, the maximum number of nodes (representing each a distinct phrase in the parse) in a LZWA trie is the same as the number attained with a binary string of length n. Clearly, the value n corresponds to the sum of path lengths from the root to all nodes of a trie, the nodes of which represent each a distinct phrase. Our plan is hence to compare the number of nodes of two trees of different *arity* but equal total path length.

We begin by recalling the notion of a *h-ary tree*, which is defined recursively as consisting of the empty tree or a root node having up to h h-ary trees as its children. Given a h-ary tree T, its *extension* is obtained by adding *leaves* in such a way that every node originally in T has now exactly h children. The following property is easily checked.

Lemma 3. *An extended h-ary tree with m internal nodes has precisely* $(h-1)m+1$ *leaves.*

Further, let the *level* of a node to be 0 for the root and 1 plus the level of the father otherwise. Now, define the *external* (respectively, *internal*) *path length* of an extended h-ary tree as the sum of the levels of all leaves (resp., internal nodes), and denote them by $E(T)$ and $I(T)$, respectively. It is easy to verify that:

Lemma 4. *In any extended h-ary tree,* $E(T) = (h-1)I(T) + h \cdot m$

Finally, we recall the following property of extended h-ary trees (see, e.g., [10,13]).

Theorem 5. *With* $\ell = \lfloor \log_h[(h-1)m+1] \rfloor$, *the minimum external path length for a h-ary tree of m nodes is*

$$[m(h-1)+1]\ell - \frac{h^{\ell+1}-h}{h-1} + h \cdot m$$

Combining Theorem 5 with Lemma 4 we get the expression

$$min(I) = \frac{[m(h-1)+1]\ell}{(h-1)} + \frac{h^{\ell+1}-h}{(h-1)^2}$$

for the minimum internal path length.

The import of the above derivation to our context is as follows. In a string of length n created by the concatenation of all distinct words of length up to k as above, we have that the number $N(k)$ of distinct phrases corresponds to the number m of internal nodes, and the length $n(k)$ of the string itself is the internal path length in a tree of minimum external path length, i.e., $N(k) = m$ and $n(k) = min(I)$. This gives us a handle to compare the maximum number of distinct phrases that can be packed in n positions using alphabets of different cardinality, say, a large alphabet of size α and a small one of size β.

Upon approximating ℓ to $\log_h(h-1) + \log_h m$ we have

$$min(I) \approx \frac{[m(h-1)+1](\log_h(h-1)+\log_h m)}{(h-1)} + \frac{h(h-1)m}{(h-1)^2} = \frac{[m(h-1)+1](\log_h(h-1)+\log_h m) + hm}{(h-1)}$$

$$= m[\log_h m + \log_h(h-1)] + \frac{1}{(h-1)}(\log_h(h-1) + \log_h m + mh).$$

We see that for large $h = \alpha$ the logarithm $\log_h(h-1)$ goes to 1 and so does $h/(h-1)$, and we get

$$min_\alpha(I) = m\log_\alpha m + 2m + o(m).$$

For small $h = \beta$, that logarithm tends to 0 but $h/(h-1)$ approaches 2, thus we also get

$$min_\beta(I) = m\log_\beta m + 2m + o(m).$$

We may now force $min_\alpha(I) = min_\beta(I)$ and derive the relationship between the largest vocabularies achievable while parsing two strings of equal length but written, respectively, over a large and a small alphabet. When the length n of the input string tends to infinity so does m, and for $m \to \infty$ we may neglect smaller order terms and impose:

$$n_\alpha = min_\alpha(I) = N_\alpha \log_\alpha N_\alpha + 2N_\alpha = N_\beta \log_\beta N_\beta + 2N_\beta = min_\beta(I) = n_\beta$$

that is,

$$N_\alpha \log_\alpha N_\alpha + 2N_\alpha = N_\beta \log_\beta \alpha \log_\alpha N_\beta + 2N_\beta$$

Theorem 6. *As $n \to \infty$, for $\alpha \geqslant \beta^2$,*

$$N_\beta < N_\alpha < N_\beta \log_\beta \alpha$$

Proof. Irrespective of the sign of $(N_\alpha - N_\beta)$, we have

$$N_\beta \log_\alpha N_\beta = \frac{N_\alpha \log_\alpha N_\alpha + 2(N_\alpha - N_\beta)}{\log_\beta \alpha} < \frac{N_\alpha \log_\alpha N_\alpha + 2N_\alpha}{\log_\beta \alpha}$$

$$< \frac{1}{\log_\beta \alpha}\left[N_\alpha \log_\alpha\left(\frac{N_\alpha}{\log_\beta \alpha}\right) + 2N_\alpha + N_\alpha \log_\alpha \log_\beta \alpha)\right]$$

$$< \frac{N_\alpha}{\log_\beta \alpha}\left[\log_\alpha\left(\frac{N_\alpha}{\log_\beta \alpha}\right) + 3\right] < \frac{2N_\alpha}{\log_\beta \alpha}\log_\alpha\left(\frac{2N_\alpha}{\log_\beta \alpha}\right)$$

since $\log_\beta \alpha < \alpha$ and for $n \to \infty$ we have $\log_\alpha N_\alpha \geq 4$, whence

$$N_\beta < \frac{2N_\alpha}{\log_\beta \alpha} \quad \text{or} \quad N_\alpha > \frac{N_\beta}{2}\log_\beta \alpha$$

and $(N_\alpha - N_\beta)$ is positive for $\log_\beta \alpha \geq 2$ or $\alpha \geq \beta^2$. Under the same conditions, we have that

$$N_\beta \log_\alpha N_\beta = \frac{N_\alpha \log_\alpha N_\alpha + 2(N_\alpha - N_\beta)}{\log_\beta \alpha} > \frac{N_\alpha}{\log_\beta \alpha} \log_\alpha N_\alpha > \frac{N_\alpha}{\log_\beta \alpha} \log_\alpha \left(\frac{N_\alpha}{\log_\beta \alpha} \right)$$

whence $N_\alpha < N_\beta \log_\beta \alpha$. □

Taking the ratio of the encodings of the two strings using $t \log t$ as the number of bits that are needed to encode t phrases, we get:

$$\frac{N_\beta \log N_\beta}{N_\alpha \log N_\alpha} > \frac{N_\beta \log N_\beta}{N_\beta \log_\beta \alpha (\log N_\beta + \log \log_\beta \alpha)} = \frac{\log N_\beta}{\log_\beta \alpha (\log N_\beta + \log \log_\beta \alpha)} = \frac{1}{\log_\beta \alpha + \frac{\log \alpha \log \log_\beta \alpha}{\log N_\beta}}.$$

This formula and that of Theorem 4 compare the number of phrases and lengths of encodings for two strings endowed with the highest vocabulary "complexity" under their respective alphabets. In particular, the theorem shows that, for sufficiently distant alphabet sizes, the number of phrases achieved in this fashion under a small alphabet of size β is always smaller than that produced by large alphabet of size α. However, such savings can never reach to a dividing factor of $\log_\beta \alpha$, which, perhaps interestingly, represents the number of levels needed for a balanced β-ry tree to achieve a yield of α leaves.

5. Conclusions and plans for future work

Most previous lossy variants of ZL and related family of encoders are built around the iterated quest for best matches within an assigned fidelity. This results in algorithms that are inherently superlinear and not easy to implement and analyze. On the other hand, it is very well known (see, e.g., [9,18,19]) that any lossy scheme of low computational complexity must have the drawback that it cannot yield the minimal distortion which can be achieved by the optimal data compression algorithm specifically tailored for that case. The approach followed in this paper concentrates thus on time performance, and builds a parse in which phrases are all distinct, with a structure that is based on self-correlations of the source and dictated deterministically by the very process of parsing. The scheme is easily implemented in linear time. The analysis of performance of lossy compression by textual substitution is not easy in general and the case arising in this paper is no exception. In practice, however, companion schemata of off-line lossy and lossless motif based compression and grammatical inference for documents of various nature have been successfully tested previously [2,3,5], and the compression achieved by LZWA and its variants are encouraging. Further, variants and deeper analyses seem thus worthy of pursuit.

Acknowledgments

I am indebted to Jacob Ziv, Michal Ziv-Ukelson and the two anonymous Reviewers for their careful scrutiny of earlier drafts of this manuscript and for providing many useful comments. I am also grateful to M. Comin, M. Melucci, M. Regnier, and J. Storer for discussions and help at various stages of the preparation.

References

[2] A. Apostolico, M. Comin, L. Parida, Bridging lossy and lossless compression by Motif Pattern Discovery, Electronic Notes in Discrete Mathematics, vol. 21, 2005, pp. 219–225. Full version in: General Theory of Information Transfer and Combinatorics, vol. II of the Report on a Research Project at the ZIF (Center of interdisciplinary studies) in Bielefeld October 1, 2002–August 31, 2003, R. Ahlswede with the assistance of L. Bäumer, N. Cai (Eds.), Springer-Verlag LNCS, 4123, 2006, pp. 787–799.

[3] A. Apostolico, M. Comin, L. Parida, Motifs in Ziv–Lempel–Welch Clef, in: Proceedings of IEEE DCC Data Compression Conference, Computer Society Press, 2004, pp. 72–81.

[4] A. Apostolico, L. Parida, Incremental paradigms of motif discovery, J. Comput. Biol. 11 (1) (2004) 15–25.

[5] A. Apostolico, L. Parida, Compression and the wheel of fortune, in: Proceedings of DCC 2003, IEEE Computer Society Press, 2003, pp. 143–152.
[7] T. Berger, J.D. Gibson, Lossy source coding, IEEE Trans. Inf. Theory 44 (6) (1998) 2693–2723.
[8] T.M. Cover, J.A. Thomas, Elements of Information Theory, Wiley-Interscience, 1991.
[9] M. Garey, D. Johnson, H. Witsenhausen, The complexity of the generalized Lloyd-Max problem, IEEE Trans. Inf. Theory 28 (2) (1982) 255–256.
[10] D.E. Knuth, The Art of Computer Programming, Addison-Wesley, Reading, Mass, 1968.
[11] A. Lempel, J. Ziv, On the complexity of finite sequences, IEEE Trans. Inf. Theory 22 (1976) 75–81.
[12] D.S. Modha, Codelet parsing: quadratic-time, sequential, adaptive algorithms for lossy compression, in: Proceedings of DCC 2003, IEEE Computer Society Press, 2003, pp. 223–232.
[13] E.M. Reingold, J. Nievergelt, N. Deo, Combinatorial Algorithms: Theory and Practice, Prentice-Hall, Englewood Cliffs, 1977.
[14] I. Sadeh, On approximate string matching, in: Proceedings of DCC 1993, IEEE Computer Society Press, 1993, pp. 148–157.
[16] T.A. Welch, A technique for high-performance data compression, IEEE Comput. 17 (6) (1984) 8–19.
[17] A.D. Wyner, J. Ziv, A.J. Wyner, On the role of pattern matching in information theory, IEEE Trans. Inf. Theory 44 (6) (1998) 2045–2056.
[18] E.-h. Yang, J.C. Kieffer, On the performance of data compression algorithms based upon stringmatching, IEEE Trans. Inf. Theory 44 (1) (1998) 47–65.
[19] R. Zamir, K. Rose, Natural type selection in adaptive lossy compression, IEEE Trans. Inf. Theory IT-47 (1) (2001) 99–111.
[20] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Trans. Inf. Theory vol. IT-23 (3) (1977) 337–343.
[21] J. Ziv, A. Lempel, Compression of individual sequences via variable-rate coding, IEEE Trans. Inf. Theory 24 (5) (1978) 530–536.
[22] J. Ziv, M. Ziv-Ukelson, Personal communication (2005).