Grammatical Ziv-Lempel Compression: Achieving PPM-Class Text Compression Ratios With LZ-Class Decompression Speed

Kennon J. Conrad and Paul R. Wilson

Independent
14628 106th Ave NE
Bothell, WA 98011, USA
kennonconrad@comcast.net

Consultant
P.O. Box 142641
Austin, TX 78714, USA
theguessworks@yahoo.com

Abstract: GLZA is a free, open-source, enhanced grammar-based compressor that constructs a *low entropy grammar* amenable to entropy coding, using a greedy hill-climbing search guided by estimates of encoded string lengths; the estimates are efficiently computed incrementally during (parallelized) suffix tree construction in a batched iterative repeat replacement cycle. The grammar-coded symbol stream is further compressed by order-1 Markov modeling of trailing/leading subsymbols and *selective recency modeling*, MTF-coding only symbols that tend to recur soon. This combination results in excellent compression ratios—similar to PPMC's for small files, averaging within about five percent of PPMd's for large text files (1 MB – 10 MB)—with fast decompression on one core or two. Compression time and memory use are not dramatically higher than for similarly high-performance asymmetrical compressors of other kinds. GLZA is on the Pareto frontier for text compression ratio and decompression speed on a variety of benchmarks (LTCB, Calgary, Canterbury, Large Canterbury, Silesia, Maximum Compression, World Compression Challenge), compressing better and/or decompressing faster than its competitors (PPM, LZ77-Markov, BWT, etc.), with better compression ratios than previous grammar-based compressors such as RePair, Sequitur, Offline 3 (Greedy), Sequential/grzip, and IRR-S.

Average compression ratios in bits per byte for various corpuses:

	GLZA	IRR-S	GRZIP-2 (SEQNTL)	LZHAM	PLZMA	PPMC-896	PPMd var J.
Calgary	2.30	n/a	n/a	2.47	2.32	2.49	2.10
Calgary (Text 6)	2.22	n/a	n/a	2.65	2.48	2.44	2.05
Canterbury	2.13	2.42	2.75	2.40	2.17	2.19	1.95
Canterbury (Text 6)	2.39	2.73	3.04	2.92	2.70	2.42	2.19
Large Canterbury	1.60	n/a	1.91	1.84	1.77	2.03	1.58
Silesia	2.17	n/a	n/a	2.28	2.21	n/a	2.09
LTCB enwik8	1.64	n/a	n/a	2.00	1.94	n/a	1.71
LTCB enwik9	1.32	n/a	n/a	1.62	1.55	n/a	1.47

1. Introduction

GLZA is a free, open-source enhanced grammar-based compressor combining several distinctive features to achieve Pareto frontier performance for compression ratio vs. decompression speed for text files, compressing better and/or decompressing faster than competitors of various sorts on a variety of benchmarks.

Features of GLZA include:

- *Recursive Superstring Reduction*, a form of greedy hierarchical dictionary construction driven by estimates of entropy-coded lengths of strings and their potential replacement symbols. Multiple passes over the input are used, but each pass efficiently computes string entropies incrementally during suffix tree construction, and thousands of distinct strings may be replaced per pass.
- Parallelized suffix tree construction pipelined with subsequent processing, for faster compression on multiprocessors.
- *Grammatical Ziv-Lempel coding* (GLZ), a flexible intermediate representation which embeds dictionary entries in the main compressed stream at their points of first use. The first occurrence of a dictionary string is left *in situ* but explicitly marked with a length code indicating how many following symbols constitute a "new" string (grammar production right-hand side); "old" symbols (repeats) have a length code of 1. Dictionary indexes are implicitly communicated to the decompressor by the natural first-occurrence order, with several advantages for subsequent modeling and encoding.
- *Selective Recency Modeling*, combining conventional dictionary techniques with selective Move-to-Front coding for specific ("non-ergodic") strings that exhibit high locality of repeats.
- Instance codes indicating the global frequencies of many symbols, supporting the removal of low-frequency symbols from the dictionary after their last occurrence.
- Adaptive modeling of length codes, recency, order-1 and instance codes.

2. Background and Motivation

2.1. Grammar-Based Compression

Grammar-based compression is a form of dictionary compression in which the strings in the dictionary may be concisely encoded in terms of other strings in the dictionary—the dictionary entry for a long string may represent the string as a sequence of symbols for shorter strings.

So, for example, when compressing English text, there's likely to be a dictionary entry for " of" and another for " the", and each will be assigned a number. There is also likely to be a dictionary entry for " th", which is a common character sequence in its own right,

and for " of the", which is just a sequence of those two symbol numbers, indicating that if you concatenate " of" with " the", you'll get the string you want.

The compressor's main job is to pick *good* strings to put in the dictionary, and to parse the input in terms of those strings. The result is a sequence of dictionary references and literals describing the whole input, plus the dictionary used to parse it.

In the common arrangement we'll be discussing in this paper, the output of the compressor consists of two things:

1. A dictionary of strings encoded as above, and
2. A main stream symbols that can be decoded using that dictionary.

Each of these things is typically entropy coded—the numbers assigned to dictionary strings are translated to shorter entropy codes for frequent strings, and longer ones for less frequent strings. The symbols in the dictionary entries are entropy-coded, as well. The dictionary is thus compressed as well, first by representing strings as concatenations of other strings, and then by entropy coding the symbols used to express those concatenations. The goal is to minimize the total size of the main sequence and the dictionary.

We can think of each dictionary entry as a production in a very rudimentary kind of grammar—a so-called "straight-line" grammar (SLG). In the example of " of", " th", " the", and " of the", we can think of the relevant dictionary entries as being grammar productions. For example, if those strings are assigned integer codes 5, 14, and 27, the relevant dictionary entries (before entropy coding) look like this:

```
5: [" " "o" "f"]
...
8: [" " "t" "h"]
...
15: [8 "e"]
...
27: [5 27]
```

As the above picture suggests, using integer symbol codes allows us to use a simple array as the dictionary index, and looking up a dictionary entry can be a single indexed load instruction. (Ignoring entropy coding, which is conceptually separate.)

Abstractly, this set of dictionary entries can be thought of as implementing a set of related grammar productions:

$N_5 \rightarrow$ " " "o" "f"
$N_8 \rightarrow$ " " "t" "h"
$N_{15} \rightarrow N_8$ "e"
$N_{27} \rightarrow N_5 \ N_{15}$

To decode symbol 27, we index into the dictionary to find its list of constituent symbols, and decode those left-to-right, recursing through other dictionary entries as needed. The recursion bottoms out wherever we hit a literal symbol, which we simply write to the output buffer in the usual way. This strictly depth-first, left-to-right "unparsing" quickly reconstructs the original input string.

This simple depth-first, left-to-right recursion over the implied "parse tree" emits literal symbols as it encounters them at the leaves, quickly generating the string left-to-right.

In standard formal grammar terms, literals are called *terminal symbols* or just *terminals* (because the recursive nested structure stops there), and dictionary symbols are called *non-terminals* (because it doesn't).

It's worth noting that a straight-line grammar is not your normal kind of "context-free grammar." Each dictionary symbol corresponds to exactly one string, not a general class of strings such as noun phrases in English or for loops in C. That trivially ensures that the whole grammar can produce the one "sentence" (string) that it's supposed to—the correct decompressed output.

Because there is no grammatical abstraction (phrase categories) of the usual sort, and there is a one-to-one correspondence between non-terminals in the grammar and the strings they represent, it turns out that the straight-line "grammar" and the parse tree are the same thing.

This very weak kind of grammar (which some consider a context-free grammar in only a degenerate sense) can't capture subtle regularities—only finite-context regularities in the sense of a PPM [3] model—but it has a big saving grace: decoding it can be very simple and very fast.

As noted above, the depth-first left-to-right unpacking (of symbols defined in terms of other symbols) is itself fairly fast, but in most cases you don't actually need to do that at all. Since every occurrence of a symbol decodes to exactly the same string, every time, you can decode it once when you first encounter it, and save a pointer to the resulting uncompressed string in the dictionary entry. Subsequent occurrences of the same symbol can be decompressed by fetching the pointer and copying the string to the output. That makes decompression about as fast as LZ77 decoding, usually executing just a few instructions between string copies.

Note that however cleverly the dictionary is constructed, SLG grammar symbols can only correspond to strings that repeat *verbatim*; they can only capture literal repeats of strings of literals. The main stream typically consists of a long series of codes for mostly relatively short strings (in roughly the 2 to 30 byte range), limiting performance of the basic grammar-based coding scheme.

A grammar-based compressor alone is therefore better at concisely exploiting the regularities *within* repeating strings (i.e., nesting of other strings) than regularities in the sequence of the larger strings. Like any other simple dictionary compressor, it fails to predict that phrases that end in certain ways are likely to be followed by phrases that begin in certain ways (as PPM can).

2.2. Compression by Greedy Textual Substitution (a.k.a. Iterative Repeat Replacement)

Like some other grammar-based compressors, GLZA's (default) compression algorithm constructs a grammar by making multiple passes over the data. At each pass it chooses some "good strings", puts those in the dictionary, and replaces all of their occurrences with dictionary (non-terminal) symbols. The reduced main stream and enlarged

dictionary from each pass are used as input to the next pass, until no more "good strings" are found.

As is an option in Apostolico and Lonardi's OFF-LINE system [7], our IRR scheme is accelerated by the use of a suffix tree constructed at each pass, and a set of non-overlapping good strings is chosen. (Strings that overlap the best strings are excluded, but may be reconsidered at subsequent passes.) On multiprocessors, it is further accelerated by parallelized operations on suffix trees.

2.3. IRR Greedy Search Heuristics

Previous researchers have explored several simple and intuitively appealing scoring functions for ranking strings in an IRR framework, including Most Frequent (MF), Longest First (LF), and Most Compressive (MC):

1. Most Frequent chooses the most frequently repeating strings and replaces them, constructing a grammar mostly bottom-up (or inside-to-outside), tending to choose symbol pairs and combines those pairs with other symbols in later passes to create a hierarchy of mostly pairs.
2. Longest First chooses the longest repeating strings and replaces them, constructing a grammar top-down (or outside-to-inside), finding the outermost repeating patterns first, and then finding other repeating strings within them.
3. Most Compressive chooses the string that offers the most immediate reduction in the input size—roughly, the string whose replacement will save the most symbols, depending on both its length and its frequency.

In general, Most Compressive has been found to be the best of these three [15], and our own experiments confirm this, but it is not entirely clear why it works as well as it does. (At least not to us.)

One seemingly obvious explanation is simply that Most Compressive is greedy in a very strong sense: it always strives to get the maximum benefit at each step, and all other things being equal, that's the best thing to do. Most Compressive is just the obvious “hill-climbing” greedy strategy.

On the other hand, the search for the best final grammar—at least in terms of number of symbols irrespective of their entropy coded size—is NP-Hard. We suspect (but do not know) that there are subtler reasons why Most Compressive is a good heuristic, but likely not the best.

Recall that the greedy search is for a small final, full "grammar" (including dictionary productions and the main stream), and that early choices may be mistakes—replacing a particular string may chop the input up in awkward ways, leaving awkward hard-to-compress chunks on either side of any replacement site. Choices made in early passes may not turn out well in later ones.

Most previous IRR-based compressors using the Most Compressive heuristic have also scored strings in terms of the net *grammar symbols* (literals and dictionary symbols) replaced, without taking into account the entropy-coded sizes of those symbols. Entropy coding is effectively assumed to be a black box that just tends to make

everything even smaller, and the IRR compressor just chooses strings that reduce the number of symbols fed to that black box. This is another potential source of error in grammar construction—even if you construct the "Smallest Grammar" in that theoretical sense [13], it may not be—*and likely isn't*—the one that will give the smallest encoded file size.

Putting those considerations aside for the moment, it is interesting to consider a basic feature of mostly-top-down grammar construction in an IRR framework, for any string scoring function similar to Most Compressive. Some strings typically occur nested in many other strings, and some typically occur in only a few contexts. For example, the English phrase "People's Republic" most often occurs within one particular enclosing phrase: "The People's Republic of China". It is a rare string outside that context, and in any given text, putting "The People's Republic of China" in the dictionary may eliminate all but one occurrence of "People's Republic"—the one within the dictionary string "The People's Republic of China". Once the longer string is taken into account, it's obvious that the shorter one is not a common string *on its own*.

We can view the construction of a grammar for compression as an attempt to create weighted rules that concisely embody something like a PPM model, with its staging of models from higher to lower, like a cascade of filters; lower-order contexts only see what higher-order contexts fail to correctly predict.

We think that mostly-top-down IRR grammar construction does something roughly analogous: it tends to filter out the high-order patterns first, and only then assess the frequencies of lower-order patterns before making difficult choices between them.

We also suspect that Most Compressive is insufficiently top-down, and thus too prone to making a certain kind of mistake in its early choices of strings for deduplication. When more frequent but shorter strings are chosen over somewhat longer and less frequent strings, there is an increased risk of chopping up the input awkwardly—if it's a bad choice, it is a choice that will chop the input *in more places* (substitution sites) for less benefit in each place (size reduction at each replacement site).

2.4. Motivation

The design of GLZA was motivated largely by the forgoing observations and working hypotheses, implying that

1. IRR string selection should be guided not by reduction in discrete symbol count, a la the Smallest Grammar Problem, but by estimates of reduction in file size, taking entropy coding into account. (If you're going to hill climb, it helps to know which way is up.)
2. Greed should be tempered with a certain amount of caution: when choosing between strings with roughly comparable benefits from replacement, choose the ones that mangle the input the least, i.e., performing longer substitutions at fewer sites, and deferring the harder choices until substring statistics are more accurate.
3. For excellent compression, our SLG grammar-based compressor can be augmented with a fast, very low-order Markov model to predict across the chunks (verbatim repeating strings) parsed by the grammar.

Two other considerations played a major role:

4. Given the resemblance between the main symbol stream and an LZ-encoded stream, we realized that we could encode the dictionary *in* the main stream, simply by explicitly marking the first occurrence of a string as something to be added to the dictionary, and replacing the subsequent occurrences with a dictionary symbol that the decompressor would then understand. This has advantages for subsequent modeling and encoding, and led to the GLZ intermediate format.
5. Given the typical chunking of the input into modest-length (SLG-parsed) dictionary strings, other regularities become apparent—particularly the difference between strings that occur with relatively stable frequencies and those that tend to recur in clusters. We therefore augmented the basic grammar-based compression scheme with *selective recency modeling*, MTF-coding only those strings with a strong tendency to recur soon.

3. Recursive Superstring Reduction and Substring Statistics Correction

GLZA uses what we call *Recursive Superstring Reduction* (RSR), which is a so-called "recursive compressor," repeatedly compressing its own output until it finds no more opportunities for compression. Each pass constructs the prefix tree of the input and then analyzes the common prefixes by traversing the internal nodes to choose the best strings to replace.

GLZA attempts to choose strings that provide good profit while also trying to minimize the damage done to profitability of subsequent passes. Generally, these are repeating strings that are relatively long and occur at a rate that is high relative to the probabilities of their constituent strings. For example, "`President of the United States`" is a good string because it's fairly long and it is considerably more frequent than you'd estimate by simply multiplying the probability of "`President of`" by the probability of "`the United States`". At each pass, a non-overlapping set of candidate "best strings" is chosen, favoring the best string in cases of overlap, and discarding the other strings it overlaps. (They may be reconsidered in a subsequent pass.) The general effect is similar to repeatedly picking the one best string and replacing its occurrences, but much faster – many strings are typically replaced per pass. When the pass is complete, the compressor replaces all occurrences of each "best string" and appends dictionary entries with unique identifiers and the strings they represent to the end of the file. The use of multiple passes allows RSR to recalculate its n-gram frequency model to account for the effects of earlier passes, e.g., that many occurrences of "`President of`" and "`the United States`" have been combined into one occurrence in the new dictionary string "`President of the United States`".

A high level block diagram of GLZA's RSR algorithm is shown below.

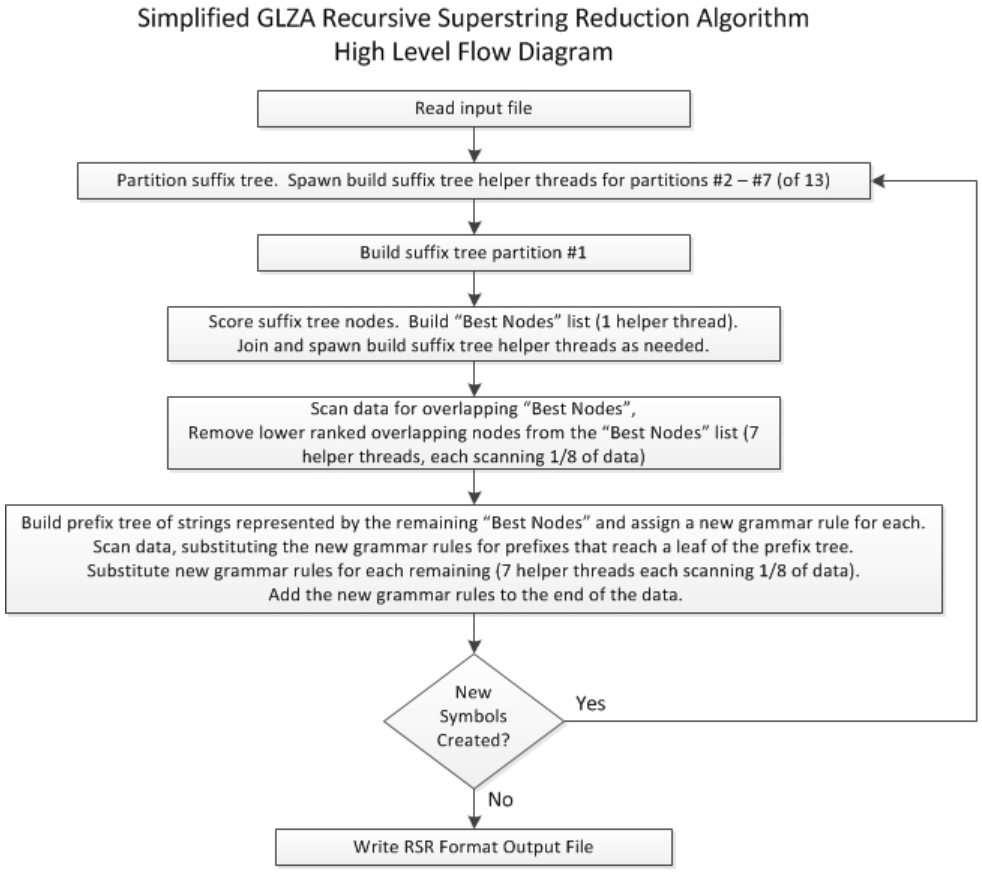

Figure 1: High Level Recursive Superstring Reduction (RSR) Flow Diagram

At each pass, a formula is used to heuristically rank the repeated strings, in term of:

N = number of input symbols
MCP = string Markov chain probability (product of observed probabilities of constituent symbols)
Repeats = number of string instances, minus 1
ProductionCost = the cost of adding a production to the grammar, default value 6.0 (bits)
ProfitRatioPower = value that sets the weight of a bias toward strings with higher profit ratios (longer strings), default value 2.0

First, we compute P_{inst}, an estimate of the immediate profit *per substitution* of choosing this string (and replacing one instance with a dictionary reference)

$$P_{inst} = Log_2(Repeats \div N) - Log_2(MCP) - 1.4$$

where 1.4 is an estimate of the cost in bits of reducing the frequencies of the constituent symbols that remain in the grammar after substitution.[1]

[1] This is a crude estimate, not a free parameter or fudge factor; the actual value is generally close to 1.4, and at most 1.44. What's being approximated here is the increase in code lengths for symbols that have some but not all of their instances eliminated—their frequencies are reduced somewhat, and their code lengths correspondingly grow a little. This is reasonably approximated by a simple constant, because only symbols that become rare grow more, and they're rare.

Then we compute P_{all}, an estimate of the total immediate profit (in bits) of replacing all repeats of this string with a dictionary reference, taking into account an estimated overhead of 7 bits for indicating the first instance of the string should be added to the dictionary:

$$P_{all} = (P_{inst} \times \text{Repeats}) - \text{ProductionCost}$$

P_{all} is basic overall profit, i.e., the net savings *in bits* due to replacing a string, after entropy coding. By itself, P_{all} would be a pretty good scoring function, embodying the Most Compressive (hill-climbing) heuristic, informed by entropy-coded sizes rather than discrete grammar symbol counts. We call this *Most Compressive, Bitwise*.

We can do somewhat better, however, by making the mostly greedy search for a grammar somewhat risk averse. In general, we prefer to replace somewhat longer and less frequent strings in favor of shorter, more frequent strings that give similar immediate compression. We therefore introduce a biasing term that weights longer strings more heavily, in a scale-independent way:

$$S = P_{all} \times (P_{inst}/\text{Log}_2(\text{MCP}))^{\text{ProfitRatioPower}}$$

If ProfitRatioPower is zero, only P_{all} matters and the score is Most Compressive, Bitwise, i.e., plain hill climbing. A value of 2 gives a modest bonus for longer strings over equally-compressive shorter ones, and that is the default we use in the results reported here. (Moderate changes to that value generally do not affect overall results significantly.)

The multipass model straightforwardly implements what we call *substring statistics correction* (SSC), which can be viewed as a (major) generalization of the *statistical substring reduction* used in statistical language models [12]. SSR only affects n-grams

that only appear in the context of larger n-grams in the model—e.g., if "`The People's Republic`" only *ever* occurs in the context of "`The People's Republic of China`", then the shorter phrase can be dropped because its frequency is implicit in the frequency of the longer one containing it. That doesn't help with phrases like

"`President of the United States`", where the constituents are used in other phrases as well.

More importantly, the longer-strings-first strategy combined with SSC has an effect much like PPM's use of higher-order models as *filters* ahead of lower-order models. (The lower-order models only see what gets past higher-order models due to their failures to predict correctly.) In RSR, this filtering strategy is used up front to construct a dictionary that can then be used with a relatively conventional LZ compressor, with negligible speed cost at decompression time.

4. GLZ Format

Recall that RSR produces two outputs: a main stream of compressed symbols, and the dictionary used to compress it. The GLZ intermediate format integrates these into a single stream, with dictionary entries interpolated into the main stream where they are first used. The first occurrence of each dictionary symbol in the main stream is replaced

with the right-hand side of the production defining it, and subsquent occurrences (repeats) refer to the corresponding dictionary entry.

Like the output of some LZ algorithms, the GLZ stream consists mainly of a sequence of alternating length codes and offset codes, but they have a different meaning. A length of 1 is used to transmit previously-seen symbols; it is followed by a dictionary offset for that symbol. A length of greater than 1 is used to transmit a "new" dictionary entry, and is followed by a sequence of that many constituent symbols for the symbol.

Independently of how the grammar was actually produced, this encodes the dictionary in the natural order of symbol occurrence, as though the first occurrence had been left *in situ* and specially marked as a pattern that will repeat later.[2]

This encoding supports productions of any length, compresses better than straightforwardly sending the dictionary separately from the main symbol sequence—there is no need to send a dictionary reference at the first occurrence of a symbol—and delays dilution of the code space until a symbol is actually used. (With the addition of a special instance count code, it can support removal of many symbols from the dictionary at their points of last use.)

The basic GLZ format is agnostic as to what kind of dictionary is actually used and what exactly an "offset" represents; GLZA currently uses LZ78-style dictionary offsets, but a compressor and decompressor could agree to use more LZ77-like window offsets or something else.

5. Grammar Characteristics

The grammar created by GLZA's RSR compressor tends to have grammar rule length and frequency distributions that are roughly Zipfian. For demonstration purposes, the grammar created by GLZA when compressing enwik9 from the Large Text Compression Benchmark (LTCB) [18] was analyzed to show these characteristics.

Figure 2 below shows the number of grammar rules as a function of the grammar rule length.

[2]This resembles Apostolico and Lonardi's "Scheme 2" encoding [7], but recursively generalized to support a hierarchical dictionary—within the first occurrence of a string, the decoder may encounter first occurrences of other strings.}

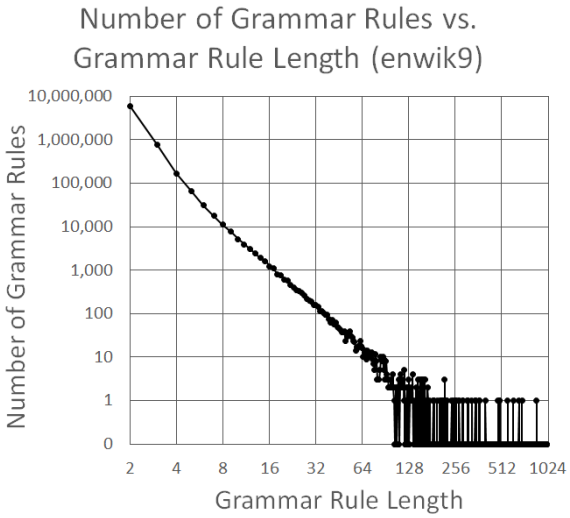

Figure 2: Number of Grammar Rules vs. Grammar Rule Length (enwik9)

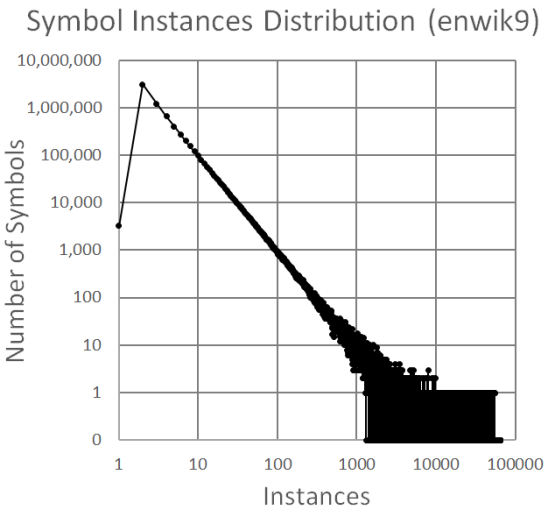

Figure 3: Symbol Instances Distribution

Approximately 84.3% of the grammar rules contain two symbols, 10.9% have three symbols, 2.3% have grammar have four symbols and the remaining 2.5% generally follow a Zipfian distribution with s=1. For highly skewed distributions like this one, the average cost of transmitting the grammar rule length is less than one bit.

Figure 3 shows the symbol instances distribution for symbols that appear up to 65,536 times. 24 symbols that are used more than 65,536 times are not shown on the plot.

The first point in the plot corresponds to 3,273 UTF-8 characters that occur once in the grammar. The rest of the points represent symbols that can be either simple or complex. The distribution of instances for symbols that occur at least twice is approximately Zipfian with s=2. This allows the cost of encoding the number of instances to be approximately two bits per new symbol, if desired.

6. GLZA Format

GLZA uses the GLZ format, but with some additions discussed below. For offset codes, GLZA used a mixture of dictionary codes and rank codes. The rank codes are used to transmit recently seen symbols using a set of fairly shallow MTF queues. The dictionary is initially empty and the initial encoding of each simple symbol is accomplished by using a special length code indicating the symbol is new and length 1 rather than the standard length 1 length code.

GLZA augments the GLZ format by using three additional fields when new symbols are sent, as follows:

- *Instances Code* – For symbols that are occur up to 15 times, this field indicates the number of instances of the symbol. For symbols that occur more than 15 times, this field provides the length of its dictionary code.

- *Ergodicity Bit* – Indicates whether a symbol is considered to be ergodic. This field follows the instances code if the symbol has more than 15 instances and has a dictionary code length of at least 11 bits. Symbols with 15 or fewer instances are assumed to be non-ergodic.
- *Space Bit* – Indicates whether a symbol is very likely to be followed by a symbol that represents a string starting with a space character (0x20). This field appears at the end of a new symbol if the symbol represents a string that ends with a space character followed by 1 or more non-space characters that do not end in the special capital encode symbol.

The length of the codes for dictionary symbol that occur up to 15 times are encoded in the file header. The instances information allows the encoder to use fairly accurate dictionary code lengths without modeling, and to remove many symbols from the dictionary when they will not be used again. The following plot shows the number of dictionary symbols created, the size of the dictionary, and the number of dictionary symbols deleted while encoding enwik9.

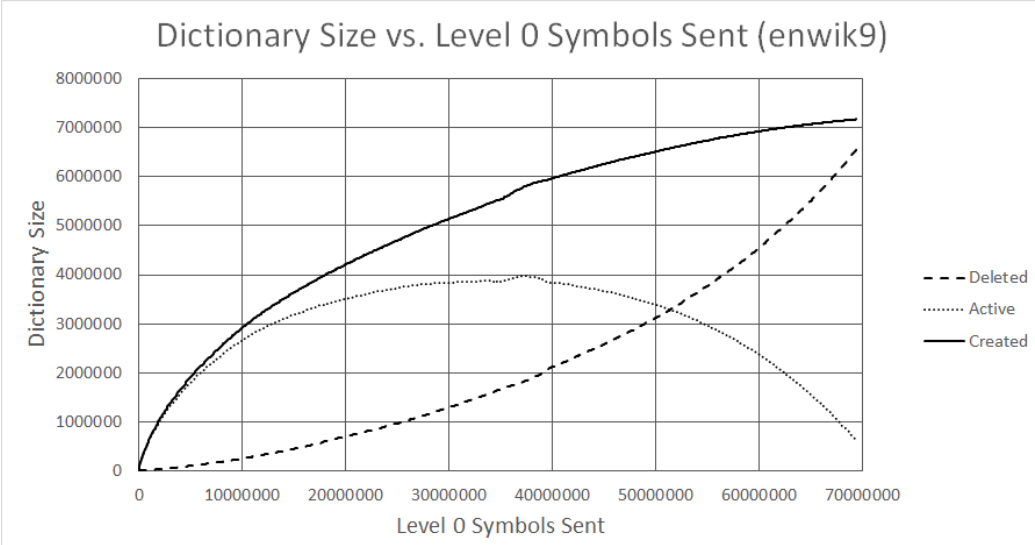

Figure 4: Dictionary Size vs. Level 0 Symbols Sent

GLZA's dictionary consists of sets of list of symbols, one list for each unique combination of the dictionary code length and the starting character of the string the symbol represents. Symbols are placed in one or more of up to 4096 code bins per starting character according to their dictionary code length, with the code bin assignments sorted from shortest code to longest code. This allows for fast conversion between dictionary symbol indexes and dictionary codes.

GLZA uses range coding to encode symbol lengths, choose between the dictionary or MTF queues, encode dictionary codes, encode MTF queue positions, encode the instances code, and to encode the ergodicity bit. Dictionary codes use a first order model to predict the starting character of the string represented by the next dictionary symbol based on the last character of the string represented by the previous symbol. All of the models used by the range coders are adaptive except the one that completes encoding of dictionary symbols after the first character has been sent.

For enwik9, approximately 8.3% of the transmitted symbols are new symbols. This allows the average cost of sending the length code to be about 0.5 bits per symbol since the vast majority of length codes are 1.

7. Results

To test the performance of GLZA relative to existing LZ-based compressors, LZ77 compressors with the highest rating (closed source and open source) on LTCB were chosen to compress the files of various standard corpora. Dmitry Shkarin's PPMd [17] was also chosen as a reference standard because it is a widely-used, high-quality, implementation of a modern PPM (with Information Inheritance). Some published results for other grammar-based compressors are included as available, as well as for PPMC-896, used as a standard for comparison in earlier studies.

Both LZ-based algorithms are variants of Igor Pavlov's LZMA, which in turn is a sophisticated variant of LZ77. PLZMA (v0.3c) is Eugene Shelwein's closed-source modified LZMA, which is interesting because its results are generally very similar to LZMA's at its maximum compression setting and it supports very large windows, so that memory limitations don't come into play. Richard Geldreich's LZHAM (v1.0) is interesting because it is optimized for high decompression speed with good ratios, like GLZA, and supports very large windows. Both LZMA variants were used with the settings that gave the best results on LTCB, which we also found to be excellent overall, after some testing.

For PPMd, we used the maximum compression settings, with a highest maximum model order (16), the largest supported model size (256 MB), and the -r1 option. (In the few very large files for which it exhausts its memory, it backs up part way and rebuilds its model before continuing, rather than just discarding it and starting from scratch. These memory-limited cases will be pointed out.)

The grammar-based compressors are SEQUITUR [5], GRZIP [16] (an improved SEQUENTIAL[9]), and Galle's IRR-S [14], an admittedly impractical and unscalable algorithm, but one whose basic strategy is most similar to GLZA's—and to our knowledge the best-compressing grammar-based compressor prior to GLZA.

The test machine’s CPU was an i7-4790K @ 4.36 GHz. Memory is Physical Memory as reported by Timer64. A RAM disk was used for the decompression tests. GLZA can decompress with either one or two threads, so times for each are listed.

Enwik9. The following table shows compression results on LTCB’s 1,000,000,000 byte enwik9 file. GLZA allows the cutoff point (estimated production cost) to be varied with the -m option, effectively allowing us to keep the less profitable productions out of the dictionary. For all other tests, we leave it at its default (realistic) value of 7, but here we also set it to 100, so that most marginally-profitable strings are left out.

Compressor	Compressed File Size (bytes)	Compression Time (sec.)	Compression Memory (MB)	Decompression Time (sec.)	Decompression Memory (MB)
GLZA v0.4	165,457,035	5,333	6,034	14.9*(2) / 20.1(1)	330*
GLZA v0.4 –m100	175,304,225	4,170	6,010	13.3(2) / 17.2(1)	86*
LZHAM	202,211,210	822	4,640	4.7	514
PLZMA	193,240,163	7,324	9,630	45.8	971
PPMd	184,909,536	376	258	381	258
PPMd –o10	183,964,893	253	258	258	258

*: Pareto Frontier

Notice that the -m100 option modestly degrades GLZA's compression ratio (by about five percent) but dramatically reduces the dictionary size and resulting memory used during decompression—by three fourths. Either way, GLZA compresses considerably better than the LZMA variants, with somewhat slower decompression than LZHAM but faster than PLZMA, and about an order of magnitude faster than PPMd, which requires about the same amount of memory for decompression as for compression.

GLZA is unusually good on this very large file, beating even PPMd on compression ratio, but that's partly due to PPMd's implementation memory limitations. (As a rule, GLZA does not beat PPMd on ratio, but comes fairly close and decompresses much faster.) Its decompression speed is intermediate between LZHAM's and PLZMA's, but with a consistently better ratio than either on text. In general we compete with LZHAM and PLZMA in terms of decompression speed and efficiency, but hope to approach PPMd in terms of ratio.

In that light, we present compression ratios for these algorithms for the Calgary Corpus [19]. PPMd results are on the right, for reference—it is not really a competitor in terms of decompression speed.

PPMd usually gets the best compression ratios, and is slow, so boldface is used for the best result among the *other* algorithms, i.e., the fast decompression competitors, and italics are used for the second-best. Cases where the PPMd results are not the best are marked with an asterisk.

We also present results for SEQUITUR, a relatively fast and effective hierarchical grammar compressor, and ppmC-896, an older PPM compressor. The results for individual files are presented in roughly their order of "textiness", for clarity of exposition, with books and papers first, binary files last, and program source files in between.

	File Size	GLZA	SEQUITUR	LZHAM	PLZMA	ppmC-896	PPMd
BOOK1	768,771	**2.29**	2.82	2.84	*2.70*	2.52	2.19
BOOK2	610,856	**1.92**	2.46	2.35	*2.20*	2.28	1.83
PAPER1	53,161	**2.41**	2.89	2.82	*2.60*	2.48	2.19
PAPER2	82,199	**2.34**	2.87	2.84	*2.65*	2.46	2.18
NEWS	377,109	**2.35**	2.85	2.67	*2.50*	2.77	2.19
BIB	111,261	**2.02**	2.48	2.39	*2.22*	2.12	1.73
PROGC	39,611	**2.47**	2.83	2.76	*2.53*	2.49	2.20
PROGL	71,646	**1.64**	1.95	1.80	*1.66*	1.87	1.44

	File Size	GLZA	SEQUITUR	LZHAM	PLZMA	ppmC-896	PPMd
PROGP	49,379	*1.70*	1.87	1.82	**1.64**	1.82	1.45
TRANS	93,695	*1.44*	1.69	1.55	**1.41**	1.75	1.22
OBJ1	21,504	3.77	3.88	*3.69*	**3.42**	3.68	*3.50**
OBJ2	246,814	2.38	2.68	*2.16*	**2.03**	2.59	2.15*
GEO	102,400	4.75	4.74	**4.21**	*4.29*	5.01	4.31*
PIC	513,216	0.75	0.90	*0.65*	**0.61**	0.98	0.75*
Average		**2.30**	2.64	2.47	*2.32*	2.49	2.09
Texty 10 (non-binary) Average		**2.06**	2.47	2.38	*2.21*	2.26	1.86
Textiest 6 Average		**2.22**	2.73	2.65	*2.48*	2.44	2.05

For the non-binary (texty and textiest) files, GLZA has a better compression ratio than lzham and SEQUITUR, and even the much slower decompressing PLZMA on the textiest. GLZA and PLZMA achieve similar compression ratios on the atypical texts (program sources and terminal transcript). For the textiest files (formatted and unformatted books and papers, news, etc.), GLZA consistently beats all the fast competitors and comes within about 8% of PPMd (Within 5 percent on the two large files.).

The following table shows compression ratios in bits per byte for the Canterbury Corpus [20]. This comparison includes compression ratios for grammar-based compressors IRR-S published in Matthias Gallé's thesis [14] and GRZIP-2 [16] (based on SEQUENTIAL [9]).

	File Size	GLZA	IRR-S	GRZIP-2 (SEQNTL)	LZHAM	PLZMA	ppmC-896	PPMd
plrabn12.txt	481,861	**2.28**	*2.73*	2.89	2.90	2.74	2.46	2.21
lcet10.txt	426,754	**1.88**	2.25	2.50	2.36	*2.23*	2.18	1.79
alice29.txt	152,089	**2.22**	2.57	2.86	2.70	*2.55*	2.30	2.04
asyoulik.txt	125,179	**2.48**	2.86	3.08	3.02	*2.85*	2.52	2.31
cp.html	24,603	**2.30**	2.64	2.95	2.75	*2.51*	2.36	2.14
xargs.1	4,227	**3.16**	3.34	3.94	3.78	*3.30*	2.98	2.87
fields.c	11,150	*2.12*	2.36	2.76	2.41	**2.12**	2.14	1.86
grammar.lsp	3,721	**2.59**	2.77	3.37	3.06	*2.65*	2.41	2.30
sum	38,240	2.52	2.86	3.17	*2.27*	**1.99**	2.71	2.33*
kennedy.xls	1,029,744	1.15	1.40	1.82	*0.47*	**0.34**	1.01	1.14*
ptt5	513,216	0.75	0.82	0.90	*0.65*	**0.61**	0.95	0.75*
Average		**2.13**	2.42	2.75	2.40	*2.17*	2.18	1.98
Texty 8 Average		**2.38**	2.69	3.04	2.87	*2.62*	2.42	2.19
Textiest 6 Average		**2.39**	2.73	3.04	2.92	*2.70*	2.47	2.23

Again, GLZA achieves better average compression ratios for the textiest files than any of the competitors, coming within about 8% of PPMd, and 5% for the large files. It achieves better compression than the other grammar compressors, beating IRR-S by an average of 12% and GRZIP-2 by an average of 22.5%.

The following table shows compression ratios in bits per byte for the Large Canterbury Corpus.

	File Size	GLZA	GRZIP-2 (SEQNTL)	LZHAM	PLZMA	ppmC-896	PPMd
bible.txt	4,047,392	**1.42**	1.86	1.79	*1.73*	1.89	1.41
E.coli	4,638,690	**2.02**	2.04	2.07	*2.04*	1.97	2.02*
world192.txt	2,473,400	**1.36**	1.84	1.65	*1.53*	2.23	1.21
Average		**1.60**	1.91	1.84	*1.77*	2.03	1.55

GLZA achieves better compression ratios for the each of the files than GRZIP-2, LZHAM or PLZMA and comes within 3% of PPMd's compression ratios.

The following table shows compression ratios in bits per byte for world95.txt from Project Gutenberg [21], the Chinese book 25393-0.txt (Shi Gong Chuan) from Project Gutenberg, website log file fp.log from Maximum Compression [22], and enwik8 from LTCB [18]. The Sequitur results were obtained with Roberto Maglica's windows port.

	File Size	GLZA	SEQUITUR	LZHAM	PLZMA	PPMd
world95.txt	2,988,578	**1.31**	1.85	1.61	*1.48*	1.18
25393-0.txt	3,573,492	**2.12**	*2.42*	2.54	2.53	2.16*
fp.log	20,617,071	**0.204**	0.339	0.282	*0.261*	0.226*
enwik8	100,000,000	**1.64**	2.29	2.00	*1.94*	1.72*

GLZA again achieves better compression ratios than the competition and has a better compression ratio than PPMd on three of the files.[3]

The following table shows compression ratios in bits per byte for the Silesia Corpus. [23]. (PPMd is somewhat memory-limited on webster, samba, and mozilla.)

	File Size	GLZA	LZHAM	PLZMA	PPMd
dickens	10,192,446	**1.82**	2.24	*2.19*	1.78
webster	41,458,703	**1.29**	1.62	*1.58*	1.24
xml	5,345,280	**0.60**	0.72	*0.63*	0.57
samba	21,606,400	*1.43*	1.46	**1.35**	1.38*
reymont	6,627,202	**1.28**	1.63	*1.56*	1.23
nci	33,553,445	0.36	*0.35*	**0.33**	0.44*
osdb	10,085,684	**2.05**	2.35	*2.21*	1.87
mozilla	51,220,480	2.39	*2.21*	**2.12**	2.44*
ooffice	6,152,192	3.31	*3.26*	**3.09**	3.24*
x-ray	8,474,240	**4.06**	*4.12*	4.13	3.62
mr	9,970,564	**2.17**	2.24	*2.21*	1.86
sao	7,251,944	5.30	**5.13**	*5.16*	5.25*
Average		**2.17**	2.28	*2.21*	2.08
Texty 6 Average		**1.13**	1.34	*1.27*	1.11
Textiest 2 Average		**1.56**	1.93	*1.89*	1.51

[3]PPM performance is again degraded somewhat on enwik8 (but not the others) due to memory limitations.

GLZA uses a special (automatically invoked) strided-difference transform on x-ray and mr; when strong short strides of numerically similar bytes are detected in a prepass, the bytes are recoded as the difference from the byte one stride length back before compressing as usual.[4] This incurs small decompression time overhead where it is used, and none where it isn't—there is no generally-applied postprocessor that adds continual overhead to decompression, only the selective inversion of a selectively-applied (and worthwhile) transform.

The above results demonstrate that GLZA tends to compress text files well, especially large text files. We believe it outperforms LZ77 by exploiting relatively fixed frequencies of common phrases and local repetition of subject term phrases and/or formatting tags/commands and idiosyncratic local punctuation. It is not highly tuned to English text or the Latin alphabet, or Western orthographies–it works well even for Mandarin Chinese.

GLZA v0.2b (generally slightly slower decompression and less efficient) was rated #1 in text decompression efficiency by World Compression Challenge 2015 [24].

8. Conclusions and Future Work

GLZA provides Pareto-frontier text compression ratios and decompression speeds, at some cost in time and memory during decompression—and performance for binary data is good as well.

In the future, we intend to improve performance further, especially for binary files, detecting and exploiting more regularities up front[5] at little cost in decompression speed, and essentially none where such regularities are *not* detected [25].

[4]This incurs small decompression time overhead where it is used, and none where it isn't—there is no generally-applied postprocessor that adds continual overhead to decompression, only the selective inversion of a selectively-applied (and worthwhile) transform.

[5]Such as the records structure of kennedy.xls [cite], which LZMA exploits but GLZA currently doesn't.}

References

[1] Ryabko, B. Ya, Data Compression By Means of a "Book Stack", Problems of Information Transmission, 1980, V. 16: (4), pp. 265-269.

[2] J. L. Bentley, D.D. Sleator, R. E. Tarjan, V. K. Wei, A Locally Adaptive Data Compression Scheme, Communications of the ACM – Vol. 29, No. 4, 1986

[3] Timothy Bell, John Cleary, and Ian H Witten. Text Compression. Prentice Hall, 1990.

[4] Craig G Nevill-Manning, Ian H Witten, and David Maulsby. Compression by induction of hierarchical grammars. In Data Compression Conference, pages 244–253, 1994.

[5] Nevill-Manning, C. G. & Witten, I. H., Compression and explanation using hierarchical grammar, Computer Journal 40(2/3), pp. 103-116

[6] Nevill-Manning, C. G. & Witten, I. H., Online and Offline Heuristics for Inferring Hierarchies of Repetitions in Sequences., Proceedings of the IEEE, 88(11), pp. 1745-1755

[7] Alberto Apostolico and Stefano Lonardi. Off-line compression by greedy textual substitution. Proceedings of the IEEE, 88:1733–1744, January 2000.

[8] John C. Kieffer and En-hui Yang, Grammar-Based Codes: A New Class of Universal Lossless Source Codes, IEEE Transactions on Information Theory, Vol. 46, No. 3, May 2000.

[9] En-Hui Yang and John C Kieffer. Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform. 1: Without context models. IEEE Transactions on Information Theory, 46(3):755–777, May 2000.

[10] N Jesper Larsson and Alistair Moffat. Off-line dictionary-based compression. Proceedings of the IEEE, 88(11):1722–1732, November 2000.

[11] En-Hui Yang and John C Kieffer. Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform. 2: With context models. IEEE Transactions on Information Theory, 49(11): 2874–2894, November 2003.

[12] L. Zhang and J. Hu, Statistical Substring Reduction in Linear Time, In Proceedings of IJCNLP2004, Sanya, Hai Nan Island, China

[13] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, April Rasala, Amit Sahai, and abhi shelat. The smallest grammar problem. IEEE Transactions on Information Theory, 51(7):2554–2576, July 2005.

[14] Matthias Gallé. Searching for Compact Hierarchical Structures in DNA by means of the Smallest Grammar Problem. Université Rennes 1, 2011.

[15] Rafael Carraascosa, François Coste, Matthias Gallé, Gabriel Infante-Lopez, Choosing Word Occurrences for the Smallest Grammar Problem. Grupo de Procesamiento de Lenguaje Natural Universidad Nacional de Córdoba, Argentina.

[16] Martin Bokler, Eric Hildebrandt, Rule reduction–A method to improve the compression performance of grammatical compression algorithms. Int. J. Electron. Commun. (AEÜ) 65 (2011) 239 – 243.

[17] D.A. Shkarin. Improving the Efficiency of the PPM algorithm. Problems of Information Transmission, Vol. 37, No. 3, 2001, pp. 226–235.

[18] Large Text Compression Benchmark (LTCB): http://mattmahoney.net/dc/text.html

[19] Calgary Corpus: http://corpus.canterbury.ac.nz/descriptions/#calgary

[20] Ross Arnold and Tim Bell. A corpus for the evaluation of lossless compression algorithms. In Data Compression Conference, pages 201–211, Washington, DC, USA, 1997. IEEE Computer Society.

[21] Project Gutenberg: http://www.gutenberg.org/

[22] Maximum Compression: http://www.maximumcompression.com

[23] Maximum S. Deorowicz. Universal lossless data compression algorithms. Ph.D. dissertation, Silesian University of Technology, Gliwice, Poland, June 2003.

[24] World Compression Challenge 2015, Nania Francesco A. (Italy): http://heartofcomp.altervista.org/MOC/TEXT/MOCADE.htm

[25] Jürgen Abel. Record preprocessing for data compression. Proceedings of the IEEE Data Compression Conference (DCC'04), pp. 521, 2004