Master's thesis
Master's Programme in Computer Science

# Implementing, analyzing, and benchmarking the Relative Lempel-Ziv compression algorithm

Eve Kivivuori

May 29, 2023

Supervisor: Professor Simon J. Puglisi

Examiner: Dr. Leena Salmela

University of Helsinki
Faculty of Science

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

| Tiedekunta — Fakultet — Faculty | Koulutusohjelma — Utbildningsprogram — Degree programme |
|---|---|
| Faculty of Science | Master's Programme in Computer Science |

Tekijä — Författare — Author

Eve Kivivuori

Työn nimi — Arbetets titel — Title

Implementing, analyzing, and benchmarking the Relative Lempel-Ziv compression algorithm

| Työn laji — Arbetets art — Level | Aika — Datum — Month and year | Sivumäärä — Sidantal — Number of pages |
|---|---|---|
| Master's thesis | May 29, 2023 | 66 |

Tiivistelmä — Referat — Abstract

In this thesis, we discuss the Relative Lempel-Ziv (RLZ) lossless compression algorithm, our implementation of it, and the performance of RLZ in comparison to more traditional lossless compression programs such as `gzip`.

Like the LZ77 compression algorithm, the RLZ algorithm compresses its input by parsing it into a series of phrases, which are then encoded as a position+length number pair describing the location of the phrase within the text. Unlike ordinary LZ77, where these pairs refer to earlier points in the same text and thus decompression must happen sequentially, in RLZ the pairs point to an external text called the dictionary. The benefit of this approach is faster random access to the original input given its compressed form: with RLZ, we can rapidly (in linear time with respect to the compressed length of the text) begin decompression from anywhere.

With non-repetitive data, such as the text of a single book, website, or one version of a program's source code, RLZ tends to perform poorer than traditional compression methods, both in terms of compression ratio and in terms of runtime. However, with very similar or highly repetitive data, such as the entire version history of a Wikipedia article or many versions of a genome sequence assembly, RLZ can compress data better than `gzip` and approximately as well as `xz`. Dictionary selection requires care, though, as compression performance relies entirely on it.

ACM Computing Classification System (CCS):
Theory of computation → Design and analysis of algorithms → Data structures design and analysis → Data compression

Avainsanat — Nyckelord — Keywords

lossless compression, relative compression, Lempel-Ziv parsing

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

# Contents

# 1. Introduction

Lossless data compression is the method of reversibly transforming a message, file or string, to occupy less space, for purposes of more efficient storage or transmission or both. Data compression was given a mathematical treatment in the 1920s, and expanded in 1948 by Shannon's seminal paper on information theory [14]: the basis of compression is that the fundamental *entropy* of a message (the input to be compressed) is typically lower than its uncompressed, "native" representation, and that by exploiting *redundancies* in the input the output can be shortened. Information theory also shows that a message cannot be *losslessly* (completely reversibly) made smaller than its entropy—compressing a maximally-compressed message will not make it any smaller[†‡]. The closely-related theory of Kolmogorov complexity [9] additionally shows that the vast majority of messages of a given fixed length are incompressible [11, p. 117]: most messages appear random, as "noise", and there are no redundancies to remove.

The primary difficulty with compressed data is that its direct manipulation is difficult: in most circumstances a decompression operation is required before the data can be used. Many commonly used schemes have a multi-step compression process, with layers of filters to expose the redundancies in the data for further compression—an attempt to use the original data would need to undo these filters, which may be difficult to do locally without knowledge of a large portion of the compressed file. Furthermore, compressed data may be self-referential, such that decompression has to be done sequentially: decompression relies on knowing previously-decompressed text. Data that needs to both be compressed and be randomly accessed has typically been compressed in *blocks*: the file is broken up into chunks, each of those chunks is individually compressed, and random access then "only" requires decompression of a

[†] A terminological note: the "space" occupied by or the "size" of a message is mostly called its "length" from here on: multi-dimensional data, such as images or video, can be (and in practice necessarily is) unravelled into a one-dimensional string, which we exclusively deal with.

[‡] *Lossy* compression is a different matter entirely: in situations where a sound or video signal is transmitted for human consumption it is not necessary to reconstruct the original signal with perfect fidelity. As long as the signal that remains after lossy compression is "good enough", a human consumer cannot tell or does not care that compression far below the theoretical lossless limit has taken place. It functions entirely differently from lossless compression and is not discussed further.

single chunk. This is merely a mitigation of the issue rather than a solution; and while it may work decently, the tradeoff is a degraded compression ratio, since identical or almost-identical substrings that fall into different chunks cannot refer to earlier occurrences of themselves, and have to be compressed separately.

The Relative Lempel-Ziv algorithm [10, 6] treated here, on the other hand, is specifically designed for easy access to the original data from its compressed form, without requiring decompressing the entire file nor some fixed-size block. There is no filtering of the input, and the compressed data is locally contained and can be accessed in essentially any order*. References are not made to the input itself, but rather to an external string called the "dictionary": the output of the Relative Lempel-Ziv compressor instructs the decompressor what to copy from the dictionary. This strategy is particularly advantageous with datasets that consist of slight variations of some underlying data, such as different assemblies of an organism's genome, or a wiki's version history; in these cases, one version of the data works as the dictionary, and the compressed output marks only where the input changes.

However, like all algorithms, Relative Lempel-Ziv is no silver bullet and has its tradeoffs. While decompression speed is on par with more common programs like `gzip`, compression is slow. Both compression and decompression are memory-intensive, requiring large data structures to be held in memory: both must load the entire dictionary in memory, and the compressor also requires a suffix array data structure, which in practice is four or eight times larger than the dictionary. Compression performance, both in terms of compression ratio and in program runtime, is greatly affected by the dictionary: a well-constructed dictionary, tailored for the input, may provide an excellent compression ratio that exceeds those attainable by `gzip` or `xz`, while a poor choice of dictionary will easily cause the output to be larger than the input.

## 1.1 Lempel-Ziv compression algorithms

The Lempel-Ziv family of compression algorithms is a large and successful one: a good proportion of internet traffic is compressed with gzip (or its successor, brotli), and thus it is found deployed as part of every internet browser. Built-in support for ZIP files compressed with the Deflate algorithm have been included on the operating systems on all computers for about twenty years now, being present in the commercial operating systems Windows XP (2001) and Mac OS X 10.3 (2003). The now-ubiquitously-supported Portable Network Graphics lossless image format also uses Deflate as the final step of its compression scheme [16, s. 10].

*Hypothetically, the file could even be decompressed in reverse with no extra overhead with a trivial change to the decompressor, assuming the compressed file is readable in reverse.

The success of Lempel-Ziv is probably due to its simplicity: replace repeated parts of the text with a short reference pointing to the first occurrence of the repeated section. Decompression of Lempel-Ziv-compressed data is also simple and cheap: details naturally vary from variant to variant, but at its simplest (such as in `gzip`) only a small buffer with a size measured in kilobytes is needed to store a short window of previously-decompressed data. Practical implementations do add some complexity, since the references need to be stored in a binary format somehow: `gzip` uses a Huffman encoding to represent references' positions and lengths and leftover literal bytes.

### 1.1.1 Lempel-Ziv 77

The first Lempel-Ziv variant is commonly referred to as "LZ77" [17]. It uses a *sliding window* of some kilobytes in size, in which *matches* (repeated text segments) are found. Matches are output as a tuple of (distance, length) numbers: these express a "go back *distance* bytes and start copying *length* bytes from there to here" expression. Note that *length* can be larger than *distance*: this represents a repetition, whereby a segment of text is repeated until it meets the specified size*. Symbols that do not appear in the sliding window are represented by a different type of phrase, by which a literal symbol can be expressed.

The Lempel-Ziv-Storer-Szymanski algorithm [15] modifies the algorithm by outputting literal symbols instead of length-distance phrases, if the representation of the phrases would be larger than the representation of the literals.

"Deflate" is a practical implementation of the LZ77 algorithm, performing string searching with the use of a hash table and further compressing the output by applying a Huffman code to the LZ77 distance-length pairs and leftover literal symbols [7]. Deflate is also the name of the binary output format as produced by the PKZIP program, which originally implemented the Deflate algorithm [2]; there exist compressors (cf. Google's Zopfli library) that output Deflate-compatible data without using the Deflate algorithm as originally specified. The `gzip` program also implements Deflate, with its own slightly-modified binary file format [3], and is widely used on Unix-like systems and on the World Wide Web—it is quite fast to decompress, and it works well on structured textual content like webpage markup tends to be.

* An example: if the sliding window contains the text "001120", an instruction to "go back 3 characters and copy 7 characters" will instruct the decompressor to place its pointer on the second "1", then start copying symbols both to the output *and* to the end of the sliding window. Because this copying is done one character at a time, the text is recycled back into the buffer for repetition. After copying the first character, the contents of the sliding window will become "011201", then after the second character, the contents will be "112012", then "120120", "201201", "012012", "120120", and finally "201201". The text written to the output will be "1201201"—a string longer than the window.

The Brotli algorithm [1] is an improvement of `gzip`, optimized specifically for compressing internet traffic and webpage content. Among its changes is a predefined built-in dictionary of about 100 kilobytes in size. Phrases output by the LZ77 compression stage can refer directly to this predefined dictionary, and so predefined phrases that appear often in web traffic but only once per document (such as `<!DOCTYPE html>`) can be efficiently compressed from the first time they appear.

The Zstandard program also uses LZ77, but takes advantage of increased resources on modern computers by utilizing larger sliding windows (megabytes instead of kilobytes) and an additional entropy coding stage. It also has support for specifying a predefined dictionary; the compression ratio of particularly small files (on the order of a couple of kilobytes) improves greatly with an application-specific dictionary.

The Lempel-Ziv-Markov chain algorithm (LZMA), familiar from being a part of the 7-Zip archiver, combines LZ77 (with big dictionaries) with a range encoder (a form of arithmetic encoding).

### 1.1.2 Lempel-Ziv 78 and Lempel-Ziv-Welch

The next compression algorithm in the family, LZ78 [18], is conceptually similar in that it parses the text into a set of phrases that appear earlier in the text. Its implementation is different, as it builds a tree-like dictionary instead of searching in a sliding window. Phrases in the dictionary consist of a reference to an earlier dictionary entry, plus a new character. Compression happens by building the dictionary, and the output takes the form of symbols that define this dictionary; Figure 1.1 illustrates an example parsing. Decompression re-builds the dictionary from the symbols, forming the uncompressed original text by expanding the sequence of references that each phrase contains. Decompression must happen in sequence, as later phrases rely on earlier ones being defined.

The LZW algorithm modifies LZ78, by having a dictionary pre-initialized with an entry for all symbols of the alphabet, along with some other practical optimizations such as “clear codes” to empty the dictionary once it grows “too big”, with an implementation-defined criterion.

## 1.2 Current work: Relative Lempel-Ziv

Relative Lempel-Ziv (RLZ) is based on LZ77’s concept of achieving compression by replacing repetitions in the text with a short reference to their first occurrence. However, the intended application of RLZ is in *random access*: it is desired to be able to access any given position in the *uncompressed* data quickly, and so the *compressed* data

`kokkokokookokokokkokokoon`

| *i* | *(ref, char)* | Expanded | *i* | *(ref, char)* | Expanded | *i* | *(ref, char)* | Expanded |
|---|---|---|---|---|---|---|---|---|
| 1 | (0, k) | k | 4 | (2, o) | ok | 7 | (6, k) | okokk |
| 2 | (0, o) | o | 5 | (4, o) | oko | 8 | (6, o) | okoko |
| 3 | (1, k) | kk | 6 | (5, k) | okok | 9 | (2, n) | on |

`0k 0o 1k 2o 4o 5k 6k 6o 2n`

**Figure 1.1:** An example illustrating the LZ78 dictionary of the Finnish sentence "kokko kokoo koko kokko kokoon" with spaces removed. Each phrase refers to an earlier phrase (phrase 0 is an empty string); for clarity, the expanded phrase is shown, although the output only includes the reference-character pairs.

needs some additional structure. RLZ therefore relies on a dictionary-based approach, with an uncompressed dictionary supplied separately from the data to be compressed, similarly to some newer predefined-dictionary-using compressors like Zstandard and Brotli. RLZ differs from these other compressors in that RLZ compression is done *entirely* against the dictionary: the output data stream only contains references to the dictionary and, where necessary, literal symbols.

The benefit of such complete separation of dictionary from input is decompression supporting a degree of random access: the output of RLZ does not rely on a sequentially-decompressed structure. Some scanning of the compressed text is required, however, as the phrase-encoding tokens of RLZ contain length information, not position information. Thus, to decompress the symbol at position $N$, the decompressor needs to count the running total of the lengths of tokens it encounters until finding the token containing symbol $N$. However, no reading of the dictionary nor writing of decompressed output needs to happen until the correct symbol is found, so this scanning and counting is quite fast. RLZ may also, under circumstances involving large and very repetitive texts and a carefully selected dictionary, be more effective at compression than sliding-window-based LZ77 compressors.

In Chapter 2 we will describe the RLZ algorithm in detail. In Chapter 3 we will discuss dictionary selection—as RLZ is entirely dependent on pre-built dictionaries for compression, a good dictionary is critical for a decent compression ratio. Chapter 4 discusses our implementation of RLZ in the form of the `rlzparse` and `rlzunparse` programs and supporting utilities. In Chapter 5 we will show benchmarks of our RLZ implementation's performance, and discuss dictionary construction parameters in concrete terms. Finally, our concluding remarks are in Chapter 6.

# 2. Algorithm description

The Relative Lempel-Ziv algorithm (RLZ for short) *parses* the input text into a series of *phrases* that occur in the separately-given dictionary. These phrases are then encoded as *tokens*, which contain the information necessary to find the phrases in the dictionary; the output will be a string of these tokens. The decompression operation, *deparsing*, takes both the dictionary and the string of tokens, and from the tokens reconstitutes the phrases and thus the original text.

## 2.1 Conventions and basic definitions

In this chapter, we will be using the following conventions and basic definitions:

- A *string* is an ordered sequence of *symbols* from an *alphabet* (which is the set of all possible symbols). The term *character* is taken to be a synonym of *symbol*. We assume that the alphabet is finite, unchanging, and has at least two symbols[†]. Unless defined otherwise, the uppercase symbols $S$ and $D$ refer to strings.
- The symbols of a string are *indexed* by a nonnegative integer: the first symbol is at index 0 in the string, the second is at index 1, and so on. Indexing a string is denoted with square brackets: the first symbol of $S$ is indexed as $S[0]$, the second as $S[1]$, and so on.
- The *length* of a string is denoted with circumfix single vertical bars: $|S|$.
- Combining the two previous points, for a finite string, the last index is $|S| - 1$, and the last character is $S[|S| - 1]$. (This is the scheme followed in the most widely-used programming languages today.) Attempting to index a finite string with an index less than 0 or greater than $|S| - 1$ is an error.
- String *concatenation* is the operation where a new string is formed by appending one to the end of another. We will use the symbol + for concatenation, relying

[†]Alphabets that grow might be in some circumstances a useful concept, such as expanding a byte alphabet into a 16-bit alphabet, but for simplicity we will define alphabets to be static. Strings over one-symbol alphabets behave pathologically and are useless in practice.

on context to differentiate concatenation from addition. With strings $S$ and $T$, the new string $S + T$ has a length of $|S| + |T|$, and $(S + T)[|S|] = T[0]$.

- We require that the alphabet is strictly totally ordered: a symbol is only equal to itself, and we can define a transitive relation $<$: given non-equal symbols $x$ and $y$, either $x < y$ or $x > y$, and if $x < y$ and $y < z$ then $x < z$.

- A substring is a contiguous segment of a string, starting at some first index $i$ and ending at some last index $j$ of the string, and denoted as $S[i \dots j]$. It shall be that $0 \leq i \leq j \leq |S| - 1$. The symbol at $S[j]$ is part of the substring: both indices are inclusive. The length of the substring is therefore $j - i + 1$; a length-one substring has $j = i$.

- A *suffix* is a special case of a substring, where the last index is the end of the string. As shorthand, we do not write the final index of a suffix: the substring $S[i \dots]$ is implicitly assumed to have an end index of $|S| - 1$, and thus $S[i \dots] = S[i \dots |S| - 1]$. Suffixes may be defined for infinite or indefinitely-long strings, in which case the suffixes also have infinite or indefinite length*.

- Likewise, a *prefix* is a special case of a substring, where the first index is 0, such as in $S[0 \dots i]$. We do not define any special notation for prefixes.

- Strings can be compared by comparing the symbols at the same index in each string. Two strings are equal if and only if they are of equal length and the symbols at all indices are equal. Otherwise, given two strings $S$ and $T$, and the first index $i$ at which $S[i] \neq T[i]$, we say that $S < T$ if $S[i] < T[i]$ and $S > T$ otherwise†. If one of the strings is shorter than the other, but otherwise has the same symbols, then the shorter string sorts before the longer one.

- In this work, we exclusively deal with finite alphabets that are a continuous sequence of natural numbers, from 0 up to one less than the size of the alphabet. Any other alphabet, such as character data, will have a isomorphism to a numerical alphabet. (This matches computational practice: our machines deal exclusively in fixed-width numbers at the lowest levels.)

With regard to alphabets, we will often refer to the *byte alphabet*, or simply a *byte string* which has the byte alphabet as its alphabet. The byte alphabet has a size of 256, the first symbol being 0 and the last being 255. Also relevant will be the similar

* The RLZ algorithms do not require that their input is finite. Parsing or deparsing can continue as long as input is available, as it is done one symbol or token at a time.

† The string relation $S < T$ is read as "$S$ sorts before $T$", not "$S$ is less than $T$".

*16-bit*, *32-bit* and *64-bit alphabets*, with respective sizes of $2^{16}$, $2^{32}$ and $2^{64}$ symbols. In various examples for the purpose of illustration we will use the 26 unadorned Latin script letters plus the space as an alphabet, with the typical alphabetical ordering and space coming first: $\{ \sqcup < \texttt{a} < \texttt{b} < \ldots < \texttt{z} \}$.

In some figures we use the dollar sign \$ to unambiguously illustrate the end of a string. This is a typographical device: \$ is not a character in the string, nor is it a member of the string's alphabet.

## 2.2 High-level description

The RLZ algorithm is given a text $S$ and a dictionary $D$, and it will produce a *parsing* of $S$ as a series of *phrases*: $S = d_1 + d_2 + \cdots + d_n$. Each of these phrases $d_i$ is either a substring of $D$, or a literal symbol that was not found in $D$. The output consists of a string of pairs of nonnegative integers, which we call *tokens*, that describe the location of each phrase in $D$ or encode the literal symbol. Compression occurs when these tokens, together with the dictionary, occupy less space than the original text.

In practice, RLZ spends its time in a string-searching loop: at any given point, we have parsed a prefix of $S$, and are trying to find the longest prefix of the remaining text $S[k...i]$ that can be found in $D$. Once we have discovered the longest prefix that is found in $D$, we output a token that describes where in $D$ it is found, and start again, searching for another segment of $S$, until we run out of input. Our search is greedy: we will always find the longest possible phrase. This is optimal, in the sense that it produces the smallest number of phrases*.

The tokens that are output have two fields, which are both nonnegative integers: the token has a *position* and it has a *length*. The position field is the index in the dictionary that the phrase starts at, and the length field is the length of that phrase. To encode symbols that do not appear in the dictionary at all, we output a *literal token*: this token has the symbol (which is itself a number) as the position and zero as the length.

An alternative approach would have a second "position" field in the tokens: instead of giving the length of the phrase, this would give the position in the *output* that

* Proof sketch: consider a segment of text that was parsed into two phrases, encoded with the tokens $(p_a, l_a)$ and $(p_b, l_b)$; the segment has length $l_a + l_b$. Because of the greedy search algorithm, we can determine that the substring $D[p_a \ldots p_a + l_a - 1]$ is the longest prefix of the text segment that can be found in $D$. If a non-greedy search was used, then in the best case a different two-phrase factorization $\{(p_x, l_x), (p_y, l_y)\}$ (where $l_x + l_y = l_a + l_b$) would be found; additionally, $l_x \leq l_a$, because otherwise, $D[p_a \ldots p_a + l_a - 1]$ would not be the longest prefix in $D$. A case where a one-phrase factorization $\{(p_z, l_a + l_b)\}$ is found would likewise require that $D[p_a \ldots p_a + l_a - 1]$ is not the longest prefix in D.

the phrase starts at. The length of the phrase would be computed by subtracting the position-in-output field of the *next* token and the position-in-output field of the current token. This approach is certainly attractive—random access would be aided by not having to compute a running total of lengths, since that work is already done by the position-in-output field. The principal issues with this approach are the description of literal symbols and the description of the end of text; since the final token only contains the information on where the final phrase *starts*, we would not know how long it needs to be. The length information of the entire uncompressed file would then need to be stored in some type of file header, or we would need some final "sentinel" token to indicate the position of the end of the uncompressed text. For simplicity of implementation we have settled on using a simple length field.

**Algorithm 1** Relative Lempel-Ziv, high-level structure.

```
1:  function RLZ-PARSE(S, D)
2:      i ← 0
3:      while i < |S| do
4:          (position, length) ← Find the longest match between S[i...] and D
5:          if length = 0 then                    ▷ Symbol S[i] was not in D at all
6:              position ← S[i]
7:              i ← i + 1
8:          else
9:              i ← i + length
10:         end if
11:         WRITE-TOKEN-TO-OUTPUT(position, length)
12:     end while
13: end function
```

A pseudocode description of RLZ parsing is shown in Algorithm 1. The complexity, and most of the hard work, in the algorithm is hidden behind finding the longest match. This is the string-searching problem, where the task is to find the location of a string $X$ within another string $T$; specifically, we use a variant of string search, where the entirety of $X$ is (probably) not in $T$, so we search for increasingly-long prefixes of $X$ in $T$. A naïve implementation of string search has an average time complexity of $O(|X| + |T|)$ and a worst-case time complexity of $O(|X| \times |T|)$, but uses no additional storage besides $X$ and $T$ themselves. In our case, the substring to find is a prefix of the suffix* of the input string, and the string that is searched is the entirety of the dictionary, and even though we are perfectly happy with simply the longest matching

* "Prefix of a suffix" is a phrase that will occur often in this section, and it may be helpful to illustrate its meaning. The entire input string $S$ is $S[0] + S[1] + S[2] + \cdots + S[|S| - 1]$. The first phrase

substring, our inputs are quite easily on the order of millions of symbols in length, each—a naïve implementation is wasteful and infeasible.

Our recourse is preprocessing: with some extra work and an auxiliary data structure, we can turn substring search into a much more feasible operation, having a worst-case time complexity of $O(|X| \log |D|)$, and in practical cases a time complexity closer to $O(|X| + \log |D|)$. Our data structure of choice is the *suffix array*, computed for the dictionary $D$. The suffix array is a permutation of all indices $0 \dots |D| - 1$, sorted by their corresponding suffixes' lexicographical position. Figure 2.1 illustrates an example of a suffix array.

| $i$ | Suffixes |
|---|---|
| 0 | hiisi␣sihisi$ |
| 1 | iisi␣sihisi$ |
| 2 | isi␣sihisi$ |
| 3 | si␣sihisi$ |
| 4 | i␣sihisi$ |
| 5 | ␣sihisi$ |
| 6 | sihisi$ |
| 7 | ihisi$ |
| 8 | hisi$ |
| 9 | isi$ |
| 10 | si$ |
| 11 | i$ |

| $i$ | Sorted suffixes |
|---|---|
| 5 | ␣sihisi$ |
| 0 | hiisi␣sihisi$ |
| 8 | hisi$ |
| 11 | i$ |
| 4 | i␣sihisi$ |
| 7 | ihisi$ |
| 1 | iisi␣sihisi$ |
| 9 | isi$ |
| 2 | isi␣sihisi$ |
| 10 | si$ |
| 3 | si␣sihisi$ |
| 6 | sihisi$ |

| $S[i]$ | h | i | i | s | i | ␣ | s | i | h | i | s | i |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| $SA[i]$ | 5 | 0 | 8 | 11 | 4 | 7 | 1 | 9 | 2 | 10 | 3 | 6 |
| Suffix | ␣sihisi$ | hiisi␣sihisi$ | hisi$ | i$ | i␣sihisi$ | ihisi$ | iisi␣sihisi$ | isi$ | isi␣sihisi$ | si$ | si␣sihisi$ | sihisi$ |

**Figure 2.1:** The suffixes of the string "`hiisi␣sihisi`".

The utility of the suffix array stems from it providing an *ordered* listing of all the suffixes of $D$. We wish to find the location of a string $S[i \dots]$ in $D$, and a good start is to find all those locations in $D$ where suffixes that begin with $S[i]$ start. Because we have, with the suffix array, an ordered listing of suffixes, we can perform binary searches to find all those suffixes in $D$ that have $S[i]$ as their first symbol. We can then perform further binary searches within that range of suffixes, finding those suffixes that have $S[i + 1]$ as second character, and so on, honing in on the suffix* $D[k \dots]$ that equals $S[i \dots]$ for the longest distance. The suffixes will eventually stop being

we parse will be a prefix $S[0 \dots i]$, and so the remaining string is a suffix $S[i + 1 \dots]$. Further parsing will result in phrases $S[i + 1 \dots j]$, $S[j + 1 \dots k]$ and so on: at each iteration of the parsing loop, the remaining string is a suffix $S[k \dots]$. When we perform string search, we do it on a growing *prefix* of this remaining suffix: we first search for the 1-long substring $S[k \dots k]$, then if we find that, the 2-long substring $S[k \dots k + 1]$, then $S[k \dots k + 2]$, and so on; thus, "prefix of a suffix".

*Or potentially a range of equally-well-matching suffixes; it does not matter which one is picked.

equal after some $l$ symbols, but, by using a sequence of fast binary searches, we have discovered that $S[i \ldots i+l] = D[k \ldots k+l]$.

Other substring-indexing data structures exist, but the suffix array is quick to construct and particularly easy to search in, and a simple search algorithm is guaranteed to find the longest substring available. However, the suffix array also uses a lot of memory—the theoretical usage is $|D|\lceil \log |D| \rceil$ bits, but in practice it is either $32|D|$ or $64|D|$ bits. It also mirrors redundancies in the dictionary: dictionary selection is discussed in Chapters 3 and 5, but in general we do not want long substrings to reoccur in the dictionary. Reoccurring long substrings slow down string searching by expanding the search space. The suffix array does not carry the information that a set of suffixes are identical for the next $n$ positions, and thus the search routine cannot skip past those $n$ positions to the point where the suffixes change.

## 2.3 Detailed description

> "Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky..." — Donald Knuth, *The Art of Computer Programming*, volume 3, section 6.2.1.

A more detailed pseudocode description of RLZ is presented in Algorithms 2 (main loop), 3 (token search, which returns the next phrase in the parsing), and 4 (binary search in the suffix array). In the longest-match-searching routine, which we call *token search*, the routine reads input symbols one at a time, keeping track of how many symbols have been read during this particular tokens' search; we call this symbol count the `offset` variable. The `offset` variable will eventually end up being the length of the token (if a match is found)—it is called "offset" instead of "length" because it is used to offset an index during the search phase.

Once an input symbol has been read, the routine performs two binary searches; our implementation calls these `search-left` and `search-right`, but they could just as well be "`search-low`" and "`search-high`". These are used to shrink the *matching range* of suffixes; the matching range is that range of suffixes which match the input up to the first `offset` symbols. The matching range starts out as being the entire suffix array, from 0 to $|D| - 1$, before a single symbol of input has been read. As input is read, the left and right bounds of the range are pulled closer together by `search-left` and `search-right`. Searching ends either once the matching range's width becomes 1 (which indicates a unique match, which is then linearly scanned as far as the dictionary

matches the input*) or the binary searches† result in a not-found status (which indicates that the token is as large as it can be—the maximal substring was found in $D$). If the binary searches fail immediately at the first read symbol, we determine that that symbol is not in the dictionary, and a literal symbol is returned.

**Algorithm 2** Relative Lempel-Ziv, with more detail. See Algorithm 3 for token search.

```
1: Sentinel position = −1, sentinel length = −1
2: function RLZ-parse(S is a file stream, D and SA_D are in memory)
3:     loop
4:         (pos, len) ← Token-search(S, D, SA_D)
5:         if pos = sentinel position and len = sentinel length then
6:             Break out from loop                    ▷ End of S was encountered.
7:         else
8:             Output-token(pos, len)
9:         end if
10:    end loop
11: end function
```

## 2.4 Suffix array computation

Computation of the suffix array is a crucial part of the functioning of this algorithm. Fast, linear-time suffix array construction algorithms exist (see [12] for a survey of them); however, implementing them in a performant manner is not trivial and was out of the scope of this thesis project. We function as if we have an oracle that computes

* This linear scanning is the optional optimization that line 25 in Algorithm 3 refers to. If it is left out of the pseudocode descriptions because it is essentially a simplified copy of Algorithms 3 and 4 embedded in an if statement. Its purpose is to skip the overhead of calling `search-left` and `search-right` when all that is needed is one equality check, and if the optimization were left out, those functions will perform the same job. It is a while-loop with the condition $c = D[SA_D[\text{leftmost}]+\text{offset}]$, fetching a new $c$ and incrementing `offset` with every iteration; it also includes the end-of-file checks and housekeeping that lines 13–14 and 28–30 of Algorithm 3 do.

† In our implementation, we perform `search-left` before `search-right`; this is an arbitrary decision. If `search-left` returns "not found", we determine that the symbol is indeed not in $D$, and skip `search-right` as redundant. However, if `search-left` returns a match, but `search-right` returns "not found", we determine we are in an error condition: binary search will not fail to find a character where one has already been determined to exist, *unless the suffix array is garbage*. If `search-right` fails where `search-left` did not, our assumptions of the suffix array's sorted-ness no longer hold, and the program crashes with an error message. Practice has shown most of these crashes to be caused by endianness issues with wide-symbol data—Section 2.5 discusses this in detail.

**Algorithm 3** The token-search routine. See Algorithm 4 for Search-left.

```
1: rc ← 0
2: function TOKEN-SEARCH(S: file, D: string in memory, SA_D: suffix array)
3:     c ← S.readSymbol(), rc ← rc + 1
4:     leftmost ← 0, rightmost ← |D| − 1
5:     bestPos ← 0, bestLen = 0, partial ← False, offset ← 0
6:     while rc ≤ |S| do
7:         if rc = |S| then
8:             return (sentinel position, sentinel length)
9:         end if
10:        leftmost ← SEARCH-LEFT(S, D, SA_D, c, offset, leftmost, rightmost)
11:        if leftmost < 0 then                    ▷ Indicates no match in the range.
12:            if partial = True then              ▷ But a partial match exists: return it.
13:                S.unread(c), rc ← rc − 1        ▷ c will be read again in the next token.
14:                return (SA_D[bestPos], bestLen)
15:            else                                ▷ c ∉ D at all: return a literal.
16:                return (c, 0)
17:            end if
18:        end if
19:        rightmost ← SEARCH-RIGHT(S, D, SA_D, c, offset, leftmost, rightmost)
20:        if rightmost < 0 then                   ▷ Indicates that SA_D is invalid.
21:            Report error, then crash.
22:        else
23:            bestLen = offset + 1, bestPos = leftmost, partial = True
24:        end if
25:        An optional optimization in case of leftmost = rightmost goes here.
26:        offset ← offset + 1
27:        c ← S.readSymbol(), rc ← rc + 1
28:        if rc = S then                          ▷ S ends with this token; the next iteration
29:            return (SA_D[leftmost], offset)     ▷ will return the sentinel.
30:        end if
31:    end while
32: end function
```

▷ The $rc$ variable, "read counter", counts how many symbols have been read from $S$. It is used for end-of-file detection, assuming $|S|$ is known. It would be possible to implement this algorithm without a global $rc$, with end-of-file detected by other means, but the pseudocode here avoids an easily-overlooked off-by-one error that causes the terminating sentinel token to be returned one token too early.

**Algorithm 4** Search-left. Search-right is similar: changes are made to line 15 (where "oldLeft" becomes "oldRight"), lines 18 and 21 ("$M-1$" changes to "$M+1$" in both), and line 23 (which becomes "$L \leftarrow M+1$"). Line 6 is not changed.

```
1: function SEARCH-LEFT(S, D, SA_D, c, offset, oldLeft, oldRight)
2:     L ← oldLeft, R ← oldRight
3:     while L ≤ R do
4:         M ← ⌊(L + R) ÷ 2⌋
5:         if SA_D[M] + offset ≥ |D| then      ▷ Suffix SA_D[M] is offset symbols long,
6:             L ← M + 1                       ▷ So skip over it: $ sorts below all else.
7:             continue with next loop iteration
8:         end if
9:         c_M = D[SA_D[M] + offset]
10:        if c_M < c then
11:            L ← M + 1
12:        else if c_M > c then
13:            R ← M − 1
14:        else
15:            if M = oldLeft then
16:                return M
17:            end if
18:            if SA_D[M − 1] + offset ≥ |D| then
19:                return M
20:            end if
21:            c'_M = D[SA_D[M − 1] + offset]
22:            if c'_M = c_M then
23:                R ← M − 1
24:            else
25:                return M
26:            end if
27:        end if
28:    end while
29:    return −1                               ▷ c ∉ D[oldLeft . . . oldRight]
30: end function
```

suffix arrays for us; indeed, our implementation of RLZ does not compute the suffix array of a dictionary, asking for it to be supplied via a file as a parameter to the parser.

## 2.5 Computation of suffix arrays for wide input symbols

Commonly-available fast suffix construction programs* only accept input using the byte alphabet. This is an issue, because one of the goals of our project was writing a RLZ parser that works with 16, 32, and 64-bit alphabets. However, with a couple of extra processing steps, it is possible to derive a suffix array for wide data with a narrow suffix array computing program.

The principal observation that makes it possible to produce a suffix array for a (for instance) 32-bit-alphabet string by using an 8-bit suffix array calculator, is that when the wider symbols are interpreted as a sequence of narrower numbers, *in big-endian order*, then an ordering of suffixes of the narrow-symbol sequence will contain within it a correct ordering of the wide-symbol sequence. The suffix array calculated with the narrower sequence will naturally be larger than the wider sequence, but with a straightforward procedure the wide-sequence suffix array can be extracted. The procedure is to remove all indices that are indivisible by `sizeof(`wide symbols`)`, and divide the remaining symbols by that size. For instance, in the case of 32-bit (4-byte) symbols, we remove indices indivisible by 4 and divide those that remain by 4.

An issue we face is that our machines, at least those with the x86 architecture, are *little-endian* by default: a number such as $31415926_{\text{hex}}$ is stored as the byte sequence `26 59 41 31`. Little-endian byte ordering breaks this scheme entirely. Let us have three strings over a 32-bit alphabet that start with the symbols $14142135_{\text{hex}}$, $17320508_{\text{hex}}$, and $31415926_{\text{hex}}$. The string that starts with 17320508 will be sorted first, followed by 31415926, and finally 14142135 is sorted last: these symbols' first-stored bytes (`08`, `26`, and `35`) sort in that order. We have not found a method for extracting a suffix array for wide data when calculated with a narrow calculator. The issue is not simply a matter of the suffix array being backwards, or that we need to extract indices of the form $4n + 3$ instead of those that are $4n$. One might also assume at first that this flipping of byte order does not matter—after all, `rlzparse` runs on a little-endian machine, so would this flipping of bytes not cancel out? No: `rlzparse` reads the byte sequence `26 59 41 31` as the 32-bit symbol $31415926_{\text{hex}}$, but the suffix

*We have used Yuta Mori's `libdivsufsort` (https://github.com/y-256/libdivsufsort) for files below $2^{32}$ bytes in size, and suggest `pSAscan` (https://www.cs.helsinki.fi/group/pads/pSAscan.html) for inputs larger than that.

array calculator sorted it as if it were $26594131_{\text{hex}}$. The binary searches that `rlzparse` uses will fail, as they require correctly sorted data.

The way around this issue is to flip the endianness of wide symbols *before* they are passed to the suffix array calculator. A copy of the dictionary is created with a simple program that reverses the order of bytes within an $n$-byte sequence, and the suffix array is calculated with that copy. The byte-order-flipped copy of the dictionary is then discarded; its only use is in suffix array calculation. The correct suffix array is then extracted from the calculated suffix array, using the discard-and-divide procedure mentioned previously. We have written a pair of programs, `endflip` and `divsuffix`, to perform endianness-flipping and suffix index dividing, with the width of integers specified as a command-line parameter.

## 2.6 Output formats

The phrases that the parser parses the input into are stored as a token consisting of two numbers: position and length. The simplest output format is to write both out as fixed-width integers, in that order, with no extra headers or footers.

The choice of what width these integers are is informed by both dictionary length and alphabet size. If the dictionary is under $2^{32}$ bytes in length, then all possible positions and lengths can be stored as 32-bit unsigned integers. If the alphabet is the 32-bit alphabet or smaller, then all possible literals can also be expressed. For the wider 64-bit alphabet, it could be possible to adopt a convention where a literal is expressed as two consecutive 32-bit literal tokens; some error handling is then necessary in the decompressor to detect input where only one of these tokens is present, or then a convention of the missing token being zero is adopted. For simplicity, our implementation disallows 64-bit alphabets with 32-bit output tokens. If the dictionary's length exceeds $2^{32}$ bytes, then wider integers are also needed—our implementation doubles the integer width to 64 bits, for reasons of ease of production, reading and alignment. This wastes space: 40-bit integers can express positions and lengths up to one less than a tebibyte ($2^{40} - 1$ bytes), and 48-bit integers can express one less than 256 TiB, which for practical purposes should still be enough in the 2020s (these sizes also approach the limits of the physically addressable address space of 64-bit CPU architectures), and so anywhere from a quarter to just under a half of the bits of each number are zero.

This raw output format has its drawbacks: the user must know what width a compressed RLZ file uses* and supply that information to the `rlzunparse` program.

*If forgotten, inspecting the file with a hex editor should provide clues as to whether 32- or 64-bit integers are used.

The user must also know what dictionary an RLZ file is based on: providing the incorrect dictionary to the deparser will produce garbage output. Some of these issues can be mitigated by following a filename naming scheme, for example by using the extension `.rlz32` for 32-bit and `.rlz64` for 64-bit files, and having the dictionary have the same name but a `.dict` extension.

In practice the output of RLZ is *sparse*, in the sense that for many practical inputs (especially non-repetitive ones), parsed with practical dictionaries, the average length of the output phrases will be quite small: values that will easily fit in a single-byte integer. This means that, for the majority of tokens in the output, three out of four length-denoting bytes will be zero. Then there is the position field: for dictionaries shorter than 16.7 MB, one byte out of four position-denoting bytes will be zero; for dictionaries smaller than 65 kB, two bytes will be zero. The magnitude of the position also fluctuates: even with large dictionaries, references to the start of it would fit into a narrower integer. Thus, around half of each token will be spent on padding: bytes that carry no numerical information, being there simply to keep our tokens aligned.

Variable-width codes would compress output size, especially with small dictionaries. There are speed and code complexity trade-offs involved, especially with bit-oriented codes that may be split over byte boundaries; our processors, memories, caches and disks work well with fixed-width integers, and having to pause for bitwise operations will slow down execution. With variable-width codes one also loses the ability to calculate the position in the file of any particular token, and so indexes built for RLZ-compressed data ("datum $f$ starts at token $n$ in the compressed file") are made complicated.

At time of writing, our RLZ implementation supports two fixed-width encodings (one uses 32-bit integers, the other uses 64-bit) and two variable-width codes. The fixed-width codes, which we call `32x2` and `64x2`, store the token's components in the machine's native byte order, which is probably little-endian on modern systems, using either 8 or 16 bytes per token. The first variable-width code is plain text: it has been useful during development but with dubious real-world utility. In it, each token's numbers are written out as ASCII text, in decimal, separated by a space and terminated by a newline. (With kilobyte-sized dictionaries this textual format has been more space-efficient than the 32-bit format.) The second variable-width code is called `vbyte` by us; it is similar to the one defined in [13] (with modifications to support encoding zero), and identical to the "Unsigned Little-Endian Base-128" format as defined by the DWARF Debugging Standard*. In it, all integers between 0 and 127 (i.e. all those that fit into 7 bits) are represented by a single byte. For larger integers, they are split into 7-bit chunks (starting from the low-order end), and written out with one byte

*https://dwarfstd.org/

per chunk, with all but the last (most-significant) chunk having the byte's high bit set to 1. Under this scheme, all 32-bit integers are encoded with 5 bytes or fewer, and all 63-bit integers with 9 bytes or fewer; an integer with the 64th bit set (unlikely to appear in RLZ output, except for literals in 64-bit data) will spill over into to a tenth byte. Our benchmarks, which are presented in Section 5.5, show that a switch from `32x2` to `vbyte` format almost halves output size and marginally speeds up compression (probably due to the reduced amount of data written), at the expense of a slightly slower decompression speed.

## 2.7 Time complexity

In this section we investigate the computational complexity of RLZ. Let $S$ be the input text to be compressed and $D$ be the dictionary it will be compressed with; $|S|$ and $|D|$ are their lengths. We shall exclude from our analysis the time it takes to compute the suffix array of $|D|$, but prior work (see [12] for a survey) has shown that this suffix array computation is an $O(|D|)$-time operation.

The worst case for RLZ is when the dictionary is wholly unrepresentative of the input: they might share no symbols at all, and if they do, only phrases with a length of 1 can be parsed. The first iteration of token search (Algorithm 3) performs binary search (Algorithm 4) over the entirety of $|D|$, instead of in a smaller sub-range, so it has a time complexity of $O(\log |D|)$. This full-range search is done for every symbol in $S$, giving us a worst-case time complexity of $O(|S| \log |D|)$.

The best case, on the other hand, is where $S$ and $D$ are identical: the parsing will consist of only one token, $(0, |S|)$. The first iteration of the token search routine does do binary search (twice) over the entirety of $D$, so it takes $O(\log |D|)$ time. In the best case, only one suffix ($D[0 \ldots]$) out of the entire $|D|$-width search range matches the input, so no binary searches will be performed at all in the following iterations. Thus, in the absolute—unrealistic—best case, RLZ has a time complexity of $O(|S| + \log |D|)$.

With practical inputs, RLZ's runtime falls between these extremes. For the first iteration of every phrase parsed, an $O(\log |D|)$ time search takes place. The search range narrows after every iteration, but the rate at this happens is entirely dependent on the nature of the inputs. For simplicity of analysis, let us assume a reasonably rapid rate of narrowing (which is realistic, for most inputs), so that the time taken for all the searches could still be described as $O(\log |D|)$ with a rather small hidden factor. Once the search range narrows to a width of 1, the search loop becomes a linear-time scan that compares $S[x + i]$ to $D[y + i]$, until the $i$ where they are no longer equal is found. This process happens for every phrase in the parsing, so the time taken to run the entire program is $O(|S| + n \log |D|)$, $n$ being the number of phrases in the parsing.

Thus, with two inputs of the same size, we would expect the one that compresses better to also compress faster—this is what we see in practice in Chapter 5.

## 2.8 Opportunities for parallelization

Phrase search is a binary search, with no clear opportunities to parallelize. However, parsing can be parallelized, with a negligible decrease of compression ratio, by splitting the text into chunks and having each thread do phrase searches on its own chunk. The only information that parallel threads would need to share is the dictionary and the suffix array, and both of those structures are read-only after being loaded from disk into memory.

On a two-processor machine, one processor could parse the text from the start to the halfway point, and the other processor from the halfway point to the end. The two output strings of tokens would then be concatenated to produce a final string of tokens describing the entire file. This parallelly-produced output differs from the optimal, single-thread output only at the tokens that describe the halfway point: more likely than not, a single-thread search would have produced a token $(x, l)$ that describes a length of text that starts before the halfway point at some point $x$ and continues after it for some length $l$. The parallel output *might* expand this into the tokens $t_1 = (x, l')$, $t_2 = (x+l',\, l-l')$, which could be combined into the single token $(x, l)$, but one cannot assume this will always happen. It is possible that the phrases $s_1 = S[x \ldots x + l' - 1]$ and $s_2 = S[x + l' \ldots x + l - 1]$ have multiple matches in the dictionary and the tokens picked for them are different than $t_1$ and $t_2$; it is also possible that $s_2$ is a prefix of an even longer phrase $s_{2'}$ that would not be found when searching for $s_1 + s_2$, such that $|s_1| + |s_{2'}| > l$. Thus, there is no general procedure for "stitching together" the halves of a parsing produced by two threads without the size of the output being one token larger than that of a single-thread parsing.

Similarly, on an $N$-processor machine, the input would be split into $N$ equal-size portions, which are parsed simultaneously. The concatenated output of these threads would be at most $N-1$ tokens larger than the single-thread parsing. We do not foresee any real-world cases where such a minor increase in output size would be an issue.

## 2.9 Decompression

In contrast to compression, RLZ decompression (or *deparsing*) is straightforward. The dictionary is read into memory, and then the compressed data—the list of (position, length) tokens—is read, one token at a time. For each token, the deparser goes to the specified position in the dictionary, and then copies the specified length's worth of

characters to the output. If the specified length is zero, the position is instead formed into an appropriately-sized symbol (for example, shrunk from 32 to 8 bits) and output as a literal. Assuming that the machine has enough random-access memory to store the entire dictionary, the deparsing operation takes $O(|S|)$ time, $S$ being the original, uncompressed text. Like usual, $D$ is the dictionary.

For decompressing data from a random index somewhere within the text, the deparser keeps track of the current position in the output text. If the deparser has been tasked with decompressing $S[x \ldots x+n-1]$, it loads the dictionary into memory as normal, then starts reading the list of tokens—but instead of writing output, it simply adds the tokens' lengths to a variable $\Sigma$, adding 1 if the length is 0. Once it comes across the token $(k, l)$ such that $\Sigma < x$ but $\Sigma + l \geq x$, it starts writing output*. For this first token that describes data to be output, it has to do some arithmetic to avoid copying the unwanted start of the phrase (if it exists): instead of writing out $D[k \ldots k+l-1]$, it instead writes out $D[k+x-\Sigma \ldots k+l-1]$. Subsequent tokens cause output to be written as normal, until arriving at the final token $(k', l')$ where $\Sigma < x+n-1$ but $\Sigma + l' \geq x+n-1$. A similar arithmetic operation is then used to avoid writing too much output.

Deparsing a substring of $l$ symbols starting at a random location $x$ is not immediate, because the deparser does not know *a priori* which phrase the location $x$ is within. The deparser must read and scan through some number of tokens and add up the lengths to learn this information, and how many tokens this is depends entirely on how compressed the data is: the higher the *mean phrase length*, the faster the seeking of $x$ is. We will write the mean phrase length as $\lambda$, and we approximate the number of tokens that must be read and summed before $x$ is found as $x/\lambda$; to this is added the time to write $l$ symbols from $D$ to output. The time complexity of decompression from a random location is therefore $O(l + x/\lambda)$.

*Or, in the case that $l = 0$ and $\Sigma = x - 1$, it simply writes out $k$: the first symbol of output was in a literal phrase.

# 3. Dictionary selection

A good dictionary is critical for an adequate compression ratio. In some situations a good dictionary is found naturally: for instance in a versioning application, where all files are approximately same size and differences from version to version are minor, a small selection of files will work well as a dictionary. On the other hand, in a situation where a single file is compressed, the choice of dictionary becomes difficult: some portion of the input must be selected, but selecting wrong will cause poor performance.

The most important parameter there is when it comes to choice of dictionary is size: it is clear that having the entire file as the dictionary will produce only a single token of output (start at zero, copy everything), whereas a single-symbol dictionary is no better than expressing every symbol literally. Even with dictionaries of the same size there are differences: a poor dictionary will consist mainly of material that appears only once, while a good dictionary will work well across the entire input.

We can thus define the notion of an *optimal dictionary*: for a given dictionary size, the optimal dictionary will result in the fewest phrases in the parsing. (The optimal dictionary is not necessarily unique.) We believe that finding and proving the optimal dictionary is an intractable problem, with no exact algorithms with acceptable runtimes for any but the smallest inputs; we suggest that the problem of finding the optimal dictionary is a variant of the maximum coverage problem.

## 3.1 Random sampling

Even though finding the optimal dictionary is infeasible, finding a *good enough* dictionary need not be. It has been shown in [5] that a sufficiently large random sampling of the input can be expected to perform well as a dictionary. The random sampling algorithm presented here is implemented in our `builddict` program, and it used during performance testing in Chapter 5. It outputs fixed-length, non-overlapping substrings of the input, and uses $n$ (the number of samples), $l$ (the length of each sample), and the input text $S$ as its only parameters. It is assumed that $n \times l \leq |S|$.

In practice, this algorithm has shown to be fast and adequate—however, it has a serious flaw, in that if the size of the output $n \times l$ is above some proportion of the

**Algorithm 5** Pick $n$ random samples of length $l$ from the input string $S$

1: $A \leftarrow$ Generate a sorted list of $n$ random positions between 0 and $|S| - l$.
2: **while** $\exists\ A[i]$ s.t. $A[i+1] - A[i] < l$ **do** ▷ Samples would overlap
3: $A[i] \leftarrow$ A new random position between 0 and $|S| - l$
4: $A \leftarrow \text{SORT}(A)$
5: (Optimization: regenerate all overlaps in one pass, before sorting.)
6: **end while**
7: **for all** $p \in A$ **do**
8: Write $S[p, p + l - 1]$ to output ▷ Recall that both indices are inclusive.
9: **end for**

size of the input, then the overlap-resolving loop on line 2 will not terminate—or more exactly, because of the randomized nature of the algorithm, the probability of the loop terminating at any given iteration will fall to almost 0.

The reason for this is interesting in and of itself, as the behavior of this non-terminating property is curious and feels like a phase transition—small tweaks to either $n$ or $l$ will cause quick termination (with the sampling decided in under a second, with only a few iterations of the while-loop) to turn into non-termination (where the loop will run for hours without a solution). The transition between these two regions is quite sharp: we have not found many ($|S|$, $n$, $l$) parameter sets for which the while loop would run for, say, a minute, with most instances being either clearly very quickly terminating or clearly looping indefinitely. (We suggest that analysis of this algorithm's phase-transition–like performance would make a good homework problem for an algorithms analysis course.)

No strict boundary such as "the program will not terminate if $nl/|S| > 2/3$" can be given, because the phase-like performance is dependent on both parameters: if we ask for $n = 1$ sample of length $l = 0.9|S|$, then we immediately get a valid solution; the same goes for other small-$n$ cases such as $n = 4$, $l = 0.20|S|$. However, we have come across cases where $l$ differs only by one and termination switches from immediate-return to no-solution: with $|S| = 12262$, $n = 13$ and $l = 864$ a solution was found in under a second, while with $l = 865$ no solution was found for hours.

Alternative algorithms of course exist: instead of generating $n$ random samples, each between 0 and $|S| - l$, we could divide the range $[0, |S|)$ up into $n$ sub-ranges $[0,\ |S|/n)$, $[|S|/n,\ 2|S|/n)$, ..., and then pick a sampling position within each sub-range (randomly or deterministically). This would force the samples to be more uniformly spread out over the entire input text, which may be desirable—the sample will not be as random as with Algorithm 5 (especially where $n \times l$ is very small, such as 0.1% of $|S|$), but might be more representative of the entire text. This method does

not remove the possibility of crystallizing into infeasibility with large outputs, either: consider the case of $l = (|S|/n) - 1$, where the sample in each sub-range must be either at index 0 or index 1 in that range. For guaranteed termination, the ranges $[0,\ |S|/n - l)$, $[|S|/n,\ 2|S|/n - l)$, ... are needed.

## 3.2 Dictionary shape

For dictionaries of a given fixed size, we may speak of them having a different *shape*: if we print out the contents of our dictionary with one sample per line, what kind of shape would we get on our screen or page? The random sampling algorithm presented in this chapter produces dictionaries with a rectangular shape.

We can characterize rectangular dictionaries by their *aspect ratio*, which we define as the ratio of $n$ to $l$. A *tall* dictionary will have shorter substrings, but more of them ($n > l$). A *wide* dictionary will have longer substrings, but fewer of them ($l > n$). Between these two falls the *square* dictionary, where the length of the substrings and number of substrings is (approximately) equal ($n \approx l$). Tallness and wideness comes in degrees: a dictionary where $l = 1.05n$ is wide according to this definition, but it will behave more like a square dictionary than an very wide dictionary of $l = 1000n$ would. Because of this, it may be desirable to describe dictionaries as *square* despite not being exactly so: we do this in Chapter 5, where we cannot construct an exactly square dictionary of a fixed size (because that would require both parameters be the square root of the size, and thus in most cases irrational), so we resort to some approximation where $n = l \pm 1$. (Likewise, integer arithmetic constrains some of the very wide dictionaries we can make: in that chapter we approximate a very wide dictionary with $n = \sqrt{2}$ by including one sample of length $l$ and another of length $\lceil l \times (\sqrt{2} - 1) \rceil$.) In Chapter 5, we write aspect ratio in the form $n{:}l$, normalized so that the smaller parameter is 1; thus, a tall dictionary of $n = 1600$, $l = 100$ would have an aspect ratio of 16:1, a square dictionary of $n = l = 400$ would have an aspect ratio of 1:1, and a wide dictionary of $n = 50$, $l = 3200$ would have an aspect ratio of 1:64.

Other dictionary shapes and sampling strategies are also conceivable, such as triangular dictionaries that contain a series of samples of steadily decreasing length. However, to avoid scope creep, we have not investigated the properties of non-rectangular, non-randomly-sampled dictionaries in this work.

# 4. Implementation

We wrote a suite of RLZ tools, including a compressor (`rlzparse`) and decompressor (`rlzunparse`), a random dictionary sampler (`builddict`) some tools for manipulating wide-symbol suffix arrays (`5to4`, `5to8`, `endflip`, `divsuffix`), and a debug tool (`suffixdump`). We have named this suite, rather simply, "rlztools".

The programs were written in C++, adhering to the C++11 standard, and they have no dependencies apart from the C/C++ standard library. The wide-symbol-manipulation tools `5to4`/`5to8` and `endflip` were simple enough to warrant writing in pure C; they follow the C99 standard, and likewise require no dependencies apart from the standard library. As a result, the programs should compile on any system with the C/C++ standard library and a POSIX-like file access interface, but so far, compilation has only been done on Linux systems with the GNU Compiler Collection.

Implementation was rather straightforward. Major sources of bugs were off-by-one errors in the binary search procedures and issues caused by the x86 architecture's little-endian byte order interfering with wide-symbol suffix arrays. The RLZ programs share some code, primarily file-access code involving reading a file of bytes as a file of wider symbols. The smaller programs written in C are contained in single files. The programs `5to4` and `5to8`, which convert 40-bit unsigned integers to 32-bit and 64-bit integers respectively, are so similar that they are written in the same file, with one or the other program being compiled by toggling a C preprocessor macro. In total, the aforementioned programs are written in about 2800 lines of code, with `rlzparse.cpp` being by far the longest at just under 1000 lines.

## 4.1 Usage

Most of the programs, when run without arguments, print out a short help message. Figure 4.1 shows the message printed out by `rlzparse`. Argument syntax is GNU-like, with both short and long option names. The `endflip` and `divsuffix` programs read and write using standard input and output instead of working with files directly, and shell redirection is used to specify input and output files.

```
rlzparse: compress with the Relative Lempel-Ziv algorithm.
Usage: rlzparse [options] -i INFILE -d DICTIONARY -s SUFFIX_ARRAY [-o OUTFILE]
Input is compressed against a dictionary and a suffix array of that dictionary;
the suffix array is a list of 32-bit binary ints in machine-native byte order.
Options:
  -w, --width 8/16/32/64    Process input and dictionary as 8/16/32/64-bit
                            units; the default is 8-bit=one-byte symbols.
  -W, --sa-width 32/64      Use 32- or 64-bit integers in the suffix array.
  -f, --output-fmt 32x2/64x2/ascii/vbyte
                            Different output formats, default=32x2.
                            32x2 and 64x2 are pairs of binary integers.
                            ascii is two space-separated numbers per line.
                            vbyte is an efficient little-endian byte encoding.
With no OUTFILE specified, output is written to 'INFILE.rlz'.
Also accepted are --dictionary, --suffix-array, --output instead of -d, -s, -o.
Other options: -q/--quiet (no output under normal circumstances),
               --progress (periodically print out a progress counter)
(rlzparse version 0.8, April 2023)
```

**Figure 4.1:** The help message printed by `rlzparse` when run without arguments.

Given an input file called "`input`", the procedure to RLZ-compress and then decompress the file would happen for example like this:

1. A dictionary called `input.dict` is formed, using for example the `builddict` tool with some appropriate parameters:
   `builddict -i input -n 6000 -l 256 -o input.dict`

2. The suffix array of the dictionary is computed with a third-party tool:
   `gensa input.dict input.sa`

3. The file is compressed against the dictionary and the suffix array:
   `rlzparse -i input -d input.dict -s input.sa -o input.rlz`

4. The suffix array and uncompressed input file can now be deleted.

5. Decompression requires the RLZ-compressed file and the original dictionary:
   `rlzunparse -i input.rlz -d input.dict -o input-reconstituted`

The RLZ tools can work with wider-width multi-byte symbols easily by specifying the `--width` (`-w`) parameter, but some extra work is needed for the suffix array calculation of the wide-symbol string. (Chapter 2.5 describes the problem we face and explains the solution to it.) With "`input`" now being a file of 32-bit integers, we would sample, compress and decompress it thus:

1. We sample the dictionary, taking care to specify symbol width:
   `builddict -i input -n 6000 -l 256 -w 32 -o input.dict`

2. The byte order of the dictionary is inverted for suffix array calculation. Our symbols are 4 bytes long, so we specify 4 as the parameter:
   `endflip 4 < input.dict > input.dict.flipped`

3. The flipped dictionary's 8-bit-wide suffix array is calculated:
   `gensa input.dict.flipped input.sa.8`

4. The suffix array of 8-bit data is divided to produce the suffix array of 32-bit data:
   `divsuffix 4 < input.sa.8 > input.sa`

5. Extra files can now be removed: `rm input.dict.flipped input.sa.8`

6. All is now in place for parsing:
   `rlzparse -i input -d input.dict -s input.sa -w 32 -o input.rlz`

7. Like before, only the dictionary and RLZ-compressed data is needed for decompression, so `input` and `input.sa` can be deleted.

8. One must remember to specify symbol width during decompression also:
   `rlzunparse -i input.rlz -d input.dict -w 32 -o input-reconstituted`

Since 32-bit indices can index files only up to 4 GiB ($2^{32}$ bytes) in size, dictionaries larger than that require a larger integer width for indexing. The `pSAscan` program* can compute suffix arrays of files up to 1 TiB ($2^{40}$ bytes) in size, and it writes 40-bit (5-byte) integers as output. Our `rlzparse` tool cannot (yet) directly read in these 40-bit integers, but it does support 64-bit integers. We thus wrote the `5to8` tool to convert 40-bit integers into 64-bit integers for interoperability with `pSAscan`. Unlike the other programs of the suite, it uses no command-line parameters; it reads data from standard input and writes it to standard output, so the shell's file redirection utilities are used to connect standard input and output with files. We then use the `--sa-width 64` (`-W 64`) parameter with `rlzparse` to load in the 64-bit integers. As an additional consequence of using a large dictionary, one needs to use an output format different from the default `32x2`, as the pointer fields of `32x2` are likewise not large enough to refer to locations in the entire dictionary. The variable-byte-encoded `vbyte` format allows references to locations above $2^{32}$, without the extra redundancy of the simpler `64x2` format—even with terabyte-size dictionaries, 3 bytes out of 8 in every position field and at least as many in the length field will be zero. Thus, compression with a very big dictionary would be done thus:

*https://www.cs.helsinki.fi/group/pads/pSAscan.html

1. `builddict -i large-input -n 42000 -l 40000 -o large-input.dict`
2. Calculate the suffix array; the pSAscan program decides the output file name automatically: `psascan large-input.dict`
3. Convert the suffix array to 64-bit integers:
   `5to8 large-input.dict.sa5 large-input.sa64`
4. `rlzparse -i large-input -d large-input.dict -s large-input.sa64 --sa-width 64 -f vbyte -o large-input.rlzv`
5. (`rm large-input.dict.sa5 large-input.sa64 large-input`)
6. `rlzunparse -f vbyte -i large-input.rlzv -d large-input.dict -o large-input-reconstituted`

## 4.2 Disk and memory usage

Dictionaries of byte-width data with a size $N$ smaller than $2^{32}$ bytes require $5N$ bytes in both disk space and random-access memory during parsing. Dictionaries larger than this require much more: although `pSAscan` can run with relatively limited memory, it will still require $6N$ bytes in disk space. The conversion process from 40-bit to 64-bit integers does not require much memory at all, as it reads and writes one symbol at a time, but the conversion will cause $14N$ bytes of disk space to be used momentarily*. The 40-bit-integer file can then be removed, and disk usage will fall to $9N$ bytes; this is how much memory will also be used during parsing.

Wider symbols will require memory and disk space proportional to their length in bytes during suffix array computation, but proportional to their length in symbols during parsing. With a dictionary of fewer than $2^{29}$ 64-bit symbols, the file will take $8N$ bytes ($N$ being the length in symbols), and the suffix array fits into 32-bit symbols. The suffix array of 8-bit data will require $32N$ bytes of space, but after index division the suffix array file will only be $4N$ bytes in length—peak disk usage is $44N$ bytes, but only $12N$ bytes after computation are left on disk and will be used during parsing. With a larger dictionary, $N \geq 2^{29}$, the file itself is larger than 4 GiB, so the jump to 64-bit integers will cause more memory to be used. The suffix array computation requires $40N$ bytes of disk space, and conversion to 64-bit integers will require another $64N$ (with disk usage reaching its peak, $112N$ bytes), but after division the suffix array will

* It would be in principle possible to start removing the 40-bit-integer file while the 64-bit integer file is created, to lessen disk space usage. We have not tried this, and expect it to be difficult or "hacky" to do in practice.

be down to $8N$ bytes—as big as the dictionary itself—and if $N < 2^{32}$, the suffix array could be reduced to $4N$ bytes by converting from 64-bit to 32-bit integers, for a total dictionary-plus-suffix-array disk usage (and parse-time memory usage) of $12N$ bytes.

This heavy disk usage, 112 times the number of symbols in the worst case, could be lessened by `rlzparse` itself computing the suffix array entirely in memory. There are no plans at the moment for implementing this, as *fast* suffix array computation is not trivial, and `rlzparse`'s codebase would be greatly complicated.

# 5. Performance

In this chapter we investigate how effective RLZ is at data compression, both in terms of how large the output is in relation to the input (compression ratio or compression factor) and how long the parsing operation takes (compression time). We compare RLZ's performance against two LZ77-based compressors, `gzip` and `xz`.

Throughout this chapter, compression ratio will be defined as the ratio between output size and input size, expressed as a percentage. This approach contrasts with the "percentage saved" approach, which is 100% minus that ratio. An example: if a 1 megabyte file was compressed to 50 kilobytes, the compression ratio would be written as 5%, although 95% of the space was compressed and the file shrunk by a factor of 20. Thus, lower numbers are more desirable, and numbers above 100% are extremely undesirable, indicating negative compression: the output was larger than the input.

We will take the size of the dictionary into account when indicating compression ratios, since the RLZ-parsed output is useless without the dictionary. Thus, for example, if the uncompressed file was 1 MB[†], the dictionary was 100 kB, and the RLZ-parsed output was 5 kB, the compression ratio is $(100\text{ kB} + 5\text{ kB}) \div 1000\text{ kB} = 0.105 = 10.5\%$. This is for fairness of comparison with other data compressors, as with a sufficiently large dictionary (such as the entire input itself) the RLZ-parsed output can be made to be as little as one token.

## 5.1 Testing datasets

Testing and benchmarking has been done with the following datasets:

- `dragon`: A generated dataset over the byte alphabet, but using only two symbols. An instance of OEIS sequence A014577 (describing a fractal called the "dragon curve" or "paper-folding curve"), mapped to the letters `L` and `R`[‡]. This sequence

[†]Throughout this chapter, we will strictly notate that 1 kB=1000 bytes, 1 MB=1000 kB, and 1 GB=1000 MB, and use alternative symbols for powers of 1024: 1 KiB=1024 bytes, 1 MiB=1024 KiB=$2^{20}$ B, 1 GiB=1024 MiB=$2^{30}$ B.

[‡] A014577 is generated with the recurrence relation $a(4n) = \texttt{R}$, $a(4n+2) = \texttt{L}$, $a(2n+1) = a(n)$.

is repetitive and highly compressible*. The intention behind this input is a "best case", "friendly" input, from which good performance can be expected, while not being entirely as trivial as simple repetitions of a fixed sequence.

- Items from the Pizza&Chili Corpus text collection [4]:
  - `pc-sources`: 211 megabytes of C and Java source code files, concatenated together, from Linux kernel and GCC versions from around 2005.
  - `pc-english`: 2.2 gigabytes of Project Gutenberg e-texts, concatenated.
  - `pc-dna`: 404 megabytes of DNA sequences, from the Human Genome Project, via Project Gutenberg's publication of those genomes. The ordinary Project Gutenberg boilerplate text has been removed, as has line wrapping: there are no newlines every 60 characters like in FASTA files. Of the 404 million characters, 8728 (around 0.002%) are not one of `A, C, G, T`: there are around 1800 newlines that separate sequences, around 5300 occurrences of the character `N`, and around a couple hundred each of the characters `B, D, H, K, M, R, S, V, W, Y`†.
- Items from the Pizza&Chili Repetitive Corpus [4, under "additional material"], containing three purely artificial files, eight "pseudo-real" files, nine real files, and nine log files. More detailed descriptions are given in Section 5.4.

## 5.2 Effect of input size on compression time

We decided to test the effect that input size has on compression time by compressing the `dragon` dataset. We chose this set because it is fast to generate in limitless quantities, it has a simple definition, it is very self-similar, and it should compress well.

We formed 20 sizes of input file, following the 1–2–5 series of preferred numbers: the first was 1 kB, then 2 kB, 5 kB, 10 kB, 20 kB, and so on, up to 1 and 2 gigabytes. For each of these files, we used 5 dictionaries. Two of the dictionaries are static: a 1 kB prefix and a 1 MiB prefix of the sequence. The other three dictionaries are 1%, 2% and 5% randomly sampled square dictionaries of the files. Since for most cases the square root of the target dictionary size, $\sqrt{|D|}$, is not an integer, we used the approximations

*"Highly compressible" is an informal characterization, but it is easily shrunk into a tiny fraction of its space. A $2^{30}$-byte portion of the sequence can be manually shrunk by 99.9% with nine search and replace expressions. Gzip version 1.10 at compression level 9 will shrink it by 99.5%. `xz` (XZ Utils) version 5.2.5, also at compression level 9, will shrink it by 99.6%.

†These are allowed symbols in FASTA format files, indicating uncertainty in the sequencing: for example, `N` indicates that the DNA sequence has one of the `ACGT` nucleotides at this position, but it cannot be determined which one.

$n = \lfloor\sqrt{|D|}\rfloor - 1$ and $l = \lfloor\sqrt{|D|}\rfloor + 1$ as parameters to `builddict`. The product of these approximations is always below $|D|$ with an absolute error of at most $\sqrt{|D|}$; the largest absolute error was 7025 bytes (0.018% too small) for the 2% dictionary of the 2 GB file, and the largest proportional error was the 2% dictionary of the 1 kB file, which was calculated as 15 bytes when it should have been 20.

Figure 5.1 shows the time it took for the program to run in each of the 100 instances, on a log-log plot. The computer used for these instances had a 3.8 GHz AMD FX 4300 (from 2012) CPU and 1866 MHz DDR3 memory, running Fedora Linux 35 with kernel version 6.0.5, with no other major processes running. The variance of runtimes under around 0.1 seconds is high, and although care was taken to measure actual CPU time instead of wall-clock time, the various overheads involved in program and file loading times are a not-insignificant portion of the time indicated, and variables such as operating system-level caching and disk controller-level caching come into play.

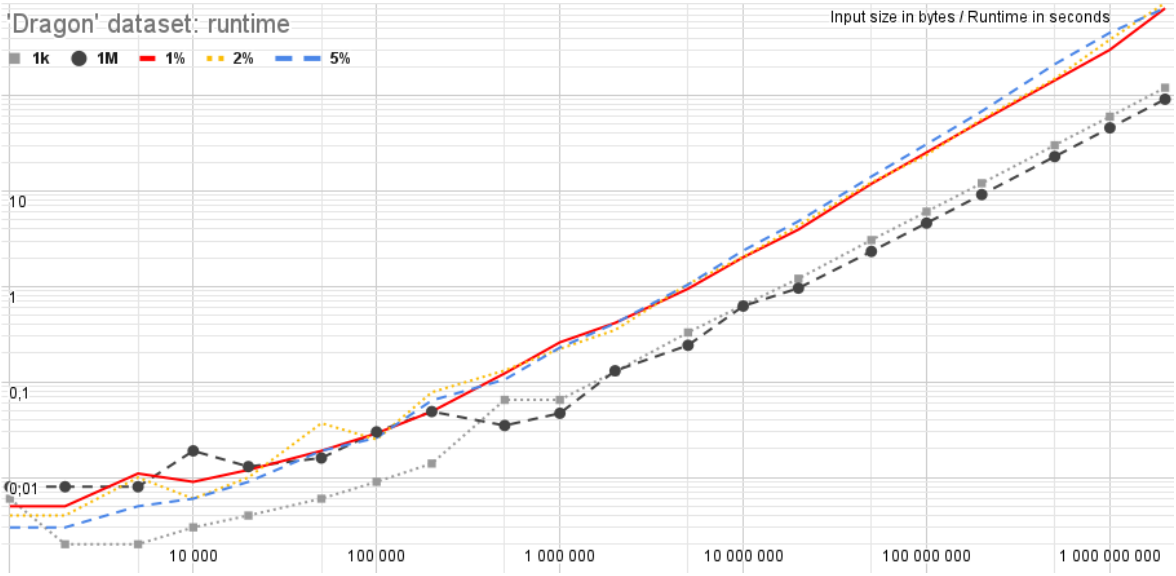


**Figure 5.1:** Run time, in seconds, plotted against `dragon` input data size, in bytes, with five different series of dictionary sizes: two of fixed length (1 kB and 1 MB) and three of fixed proportion of the input (1%, 2% and 5%). Both scales are logarithmic: the longest runtimes are on the order of 1000 seconds, or about 20 minutes.

Once the runtimes grow out of the millisecond range, and the input size grows above a megabyte, distinct populations start to form. The instances using the fixed-size one-megabyte dictionary are always the fastest, closely followed by the one-kilobyte static dictionary—looking at the underlying numbers, the 1k instances are constantly about 1.31 times slower than the 1M instances*. The fixed-size-dictionary population is then trailed by the much slower fixed-proportion, variable-size-dictionary population.

* The instances with input sizes of 2 and 10 megabytes are anomalous, with almost-identical runtimes. The 20 MB/1k instance is also only 1.25 times slower than the 20 MB/1M instance.

This population is all in all quite close in its runtimes, with the 5% dictionary being in general slightly slower than the 1% dictionary (by 1.2 to 1.5 times), and with the 2% dictionary somewhere in between. The 1% dictionary instances are slower than the 1k dictionary instances by around 3 times, but this ratio has a growing trend: by the 2 gigabyte input, the 1% dictionary is 6.7 times slower than the 1k dictionary. It is difficult to predict with confidence with a dataset of this size whether this trend holds for larger and larger inputs, but with the data we have, diverging trend lines appear to be the case: the fixed-proportion dictionaries will become slower and slower with respect to the fixed-size dictionaries. This agrees with our time complexity calculations derived in Section 2.7: larger dictionaries are generally slower, but dictionaries that are too small (and thus frequently cause token search to be started anew) will be slow again, because of the overhead involved with the first stages of token search.

Other instances of trend lines with different slopes are shown in the next two plots. Figure 5.2 plots the number of phrases output by each instance—for example, all instances with input sizes under a megabyte that used the one-megabyte static dictionary output a single phrase, since the input is a subset of the dictionary, while parsing the 1 kilobyte input instance with the 1% dictionary output 218 phrases – approximately $1000 \div 4$, since the dictionary consisted of two 4-byte samples. Figure 5.3 plots the compression ratio: the size of the output compared with the input. Here, the dictionary size is taken into account: the output size is the size of the dictionary plus the size of the tokens, with each phrase taking up eight token.

From Figure 5.2, we see that the static dictionary instances behave quite differently from the variable-size instances. For smaller inputs, the parser parses them into fewer phrases, but the 1k dictionary instance eventually parses into larger outputs than the variable-dictionary instances (the threshold is 2 MB of input for the 5% dictionary and 10 MB of input for the 1% dictionary at which they will parse smaller; the dictionaries of these instances are the same size), and it appears that the 1M instance will eventually parse larger than the variable-size instances*.

The compression ratio plot (Figure 5.3) is approximately the inverse of the number-of-phrases plot: lines trend downward. Additionally, a clear leveling-off is observed: the 1k dictionary initially starts as the best dictionary, being the *only* dictionary for a while for which the output is smaller than the input. It eventually flattens out, and never improves upon the compression ratio of 18.75%. The 5% dictionary is next seen to level off, and right at the end of the chart the 2% and 1% dictionaries

* The number of phrases when parsed against the 1M dictionary grows as fast as the input size does (in a 2, 2, 2.5× pattern), whereas all of the variable dictionaries grow geometrically with a factor of about 1.48. The output of parsing against the 1M dictionary will exceed that of the 5% dictionary at inputs of around two terabytes, and those of the 1% dictionary at around 10 terabytes.

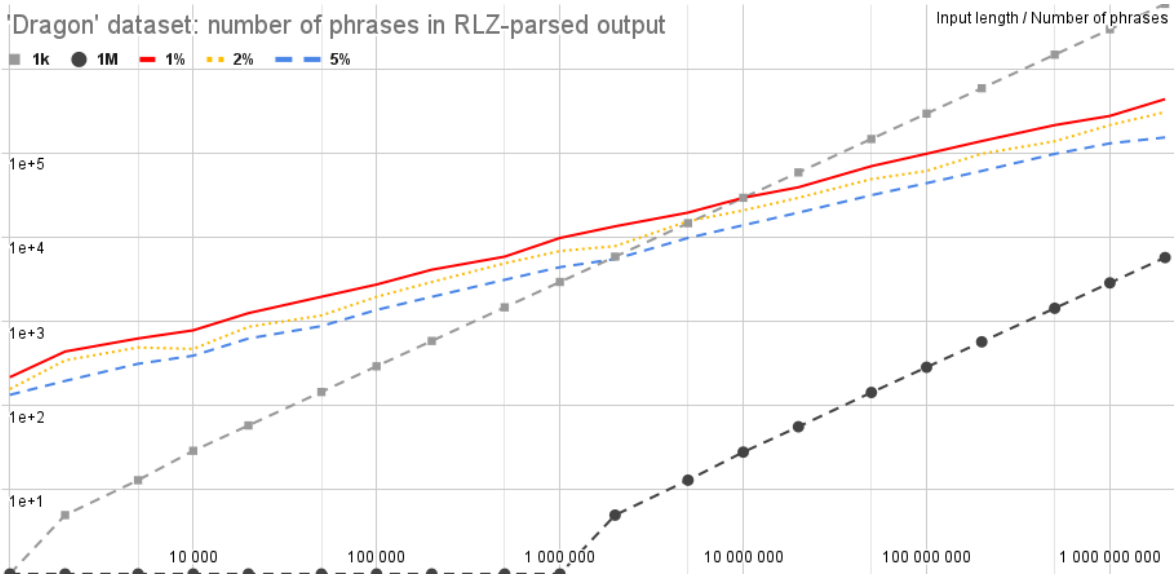


**Figure 5.2:** Number of phrases output, plotted against `dragon` input data size, with five different series of dictionary sizes. Both scales are logarithmic. The dictionaries are the same as in Figure 5.1: 1k and 1M are fixed-size dictionaries of 1 kB and 1 MB, while 1%, 2% and 5% are variable-sized dictionaries, being their respective proportion of the input size.

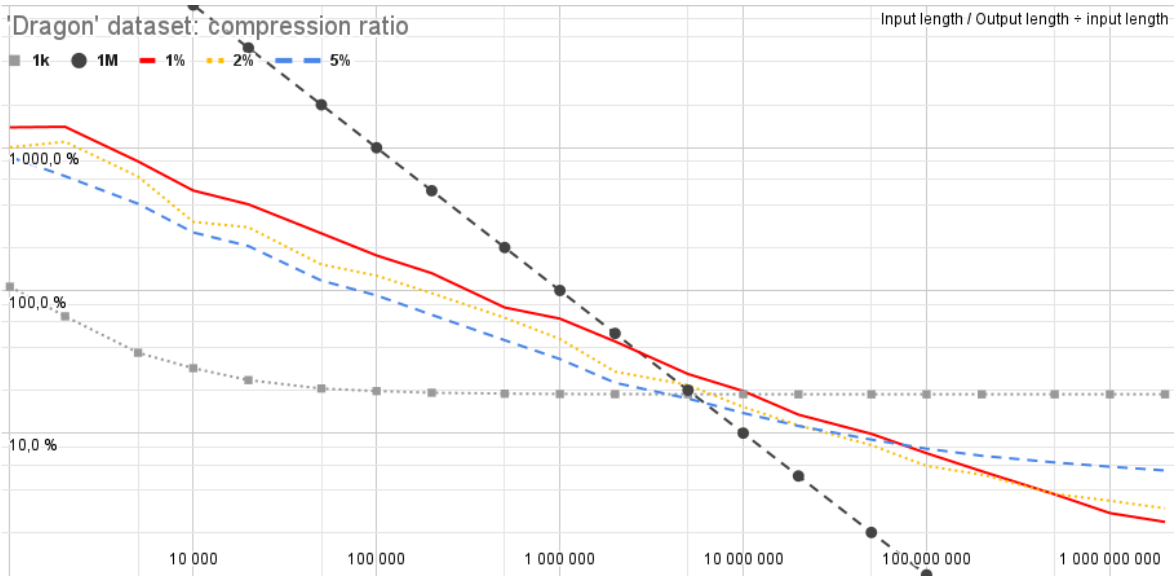


**Figure 5.3:** Compression ratio of output vs input, plotted against `dragon` input data size. There are five different series of dictionary sizes, and compression ratio takes the dictionary size into account. Each phrase is eight bytes. Both scales are logarithmic. Dictionaries are the same as in Figures 5.1 and 5.2: 1k and 1M are fixed-size, while 1%, 2% and 5% are proportional to input size.

approach their optima. The 1M dictionary, on the other hand, started off-the-chart (with the output being 1000 times longer than the input) and ended off-chart also: for the 2 GB input, it produced a mere 5721 phrases, for a total output size of 1 045 768 bytes: 0.068%.

This flattening-out of the compression ratio plot is due to the geometric growth of the number of phrases output: for the fixed-size dictionaries, the number of phrases doubles, and for the fixed-proportion ones, it grows by a factor of around 1.5. On a long-enough scale the growth in the number of phrases will catch up with, and then overtake, the growth of the dictionary, becoming the dominant term in the size of the output files.

Having analyzed these plots, and the plots in further sections, we must emphasize the artificial nature of the `dragon` input: its repetitive, fractal structure is among the best inputs that a compressor can be given. Next, we will analyze performance on real files, as well as pseudo-realistic sequences and naturally repetitive datasets, and analyze how choice of dictionary affects both runtime and compression ratio.

## 5.3 Results for real data

Initial testing on real data was done with the 211-megabyte `pc-sources` dataset, which consists of (commented) C and C-like source code of the Linux kernel and GCC compiler. This file is quite compressible by traditional, LZ77-family compressors: `gzip -9` compresses it to 47 megabytes and `xz -9` compresses it to 31 megabytes.

It was arbitrarily decided to try compression with dictionaries of 1% and 5% of the input. Two dictionaries for each size were constructed: one was a square dictionary, randomly sampled with the `builddict` program*, and the other was simply a prefix of the input.

Table 5.1 shows how long it took `rlzparse` to parse the `pc-sources` file, and how large the output was. There are three time columns: "user" time indicates the time the process spent in operating system user space, primarily performing token search; "system" time indicates the time spent in kernel space, serving system calls such as file access†. Compression performance is not good: only one of the four, the 5% square

*Because neither 1% nor 5% of the input was a square number, the parameters $n = l = 1452$ and $n = l = 3247$ were used.

† A previous version of `rlzparse` had a bug caused by an oversight, where the output file was flushed (i.e. buffers were emptied and data was forcefully written to the disk) with every write. All writes were 8 bytes long, and one was done for every token. With the extra flushes, runtime was around 160% of what it is now, and a third of that time, around a minute at each run, was spent in the kernel. Removing those extraneous flushes allowed the operating system to start using its output cache, and thus system time fell from a minute to a second or two.

| Dict. shape | Tokens output | Ratio | MPL | User time | System time | Total time |
|---|---|---|---|---|---|---|
| Prefix, 1% | 35,050,445 | 134% | 6.02 | 118.0 sec | 1.46 sec | 119.5 sec |
| Prefix, 5% | 27,234,004 | 108% | 7.74 | 180.9 sec | 1.56 sec | 182.5 sec |
| Square, 1% | 26,438,458 | 101% | 7.98 | 102.6 sec | 1.33 sec | 103.9 sec |
| Square, 5% | 19,558,060 | 79% | 10.78 | 149.1 sec | 1.28 sec | 150.4 sec |

**Table 5.1:** Runtimes and compression ratios for compressing the 211-megabyte `pc-sources` dataset. MPL stands for mean phrase length; $8 \div \text{MPL}$ is the compression ratio without the dictionary, while the "Ratio" column takes into account the size of the dictionary. Ratios over 100% indicate that the output is larger than the input.

| Dict. shape | Tokens output | Ratio | MPL | User time | System time | Total time |
|---|---|---|---|---|---|---|
| Prefix, 1% | 235,544,845 | 86.3% | 9.38 | 3159 sec | 17.5 sec | 3177 sec |
| Prefix, 5% | 184,740,373 | 71.8% | 11.96 | 4058 sec | 17.7 sec | 4076 sec |
| Square, 1% | 208,682,811 | 76.5% | 10.59 | 3057 sec | 16.0 sec | 3073 sec |
| Square, 5% | 168,541,603 | 66.0% | 13.11 | 3973 sec | 17.4 sec | 3990 sec |

**Table 5.2:** Runtimes and compression ratios for compressing the 2.2-gigabyte `pc-english` dataset.

| Dict. shape | Tokens output | Ratio | MPL | User time | System time | Total time |
|---|---|---|---|---|---|---|
| Prefix, 1% | 33,021,742 | 66.4% | 12.23 | 363.6 sec | 2.16 sec | 365.8 sec |
| Prefix, 5% | 28,083,729 | 60.6% | 14.38 | 545.3 sec | 2.21 sec | 547.5 sec |
| Square, 1% | 32,772,320 | 65.9% | 12.33 | 369.8 sec | 2.06 sec | 371.9 sec |
| Square, 5% | 28,021,791 | 60.5% | 14.41 | 548.2 sec | 2.17 sec | 550.4 sec |

**Table 5.3:** Runtimes and compression ratios for compressing the 404-megabyte `pc-dna` dataset.

dictionary, managed to produce a net saving of data, and even then only a fifth was saved.

Further testing was done on the `pc-english` and `pc-dna` datasets. The former is a 2210-megabyte collection of English-language literature from Project Gutenberg, concatenated into one file. The latter is a 404-megabyte concatenation of DNA sequences from the Human Genome Project, with a byte alphabet and no line wrapping. We formed dictionaries of 1% and 5% prefixes and squares for both datasets*. In the resultant runtimes, which are in table 5.2 for `pc-english` and 5.3 for `pc-dna`, we see the non-linearity of the algorithm: `pc-english` had an input and dictionaries about ten times bigger than `pc-sources`, but ran around 30 times slower than it, and `pc-dna` was twice as big and ran 3 (for the prefix runs) to 3.6 (for the square runs) times slower.

## 5.4 Results for repetitive data

Testing was done on all files of the Pizza&Chili Repetitive Corpus. We compressed each file with `rlzparse` four times: once with a 1 MiB prefix dictionary, and once each with 0.1%, 1%, and 5% square dictionaries. (The approximation of $n = l = \lfloor\sqrt{0.001|S|}\rfloor$, or likewise with 0.01 and 0.05, was used to define the dictionary.)

For comparison, we also compressed each file with the LZ77-utilizing compressors `gzip` and `xz`, both at their default settings and at their maximum compression level, obtained with `gzip -9`/`xz -9`. In the majority of cases† the difference in output size was negligible, on the order of an extra 0.1–1 percentage point improvement to compression ratio, so we have opted to only include here the results of those runs with the `-9` flag.

Results are presented broken down by each of the four categories of files: artificial mathematical sequences, "pseudo-real" files obtained by concatenating mutated versions of a text, real files that are inherently repetitive (such as DNA or the entire history of a Wikipedia article), and log files. Within each of the tables 5.4—5.7, the percentage refers to the compression factor (output divided by input), and for RLZ files,

* For the `pc-english` dataset the parameters $n = l = 4701$ and $n = l = 10512$ were used for the 1% and 5% square dictionaries, respectively. The `pc-dna` dataset used the parameters $n = l = 2009$ for 1% and $n = l = 4494$ for 5%.

† Exceptions: the files `BGL`, `Thunderbird` and `Windows` were around 0.05 percentage points *larger* when compressed with `xz -9` than their default-setting counterparts. On the other hand, the files `cere`, `coreutils`, `Escherichia_Coli`, and `para` were shrunk considerably more with `xz -9`: `cere` and `para` had a 22 pp improvement in compression ratio, from around 23% to 1% input-to-output; `coreutils` shrunk by almost 10 pp and `Escherichia_Coli` shrunk by 4 pp, from 9% to just under 5%. The `gzip` program was more consistent: most runs of `gzip -9` improved upon the default by between 0.2 to 1 pp, and only the `influenza` file was compressed more, shrinking by 4.3 pp.

*includes* the size of the dictionary; the size without the dictionary is easily derived*. The file size refers to the size of the compressed output itself, excluding the dictionary. The RLZ output format used was pairs of fixed-width 32-bit integers (`32x2`), so the number of phrases in the factorization can be calculated as file size divided by 8, and thus the (approximate) mean phrase length can be found.

### 5.4.1 Artificial data

The Pizza&Chili repetitive corpus had three artificial files, taking the form of binary sequences with a mathematical definition, written with one character per symbol.

**`fib41`** This is an instance, or rather a prefix, of the infinite Fibonacci word, OEIS sequence A003849, expressed with the characters "`a`" and "`b`". The definition is recursive: the base cases are $F(1) =$ `a` and $F(2) =$ `ab`, and the $n$th Fibonacci word is defined as $F(n) = F(n-1) + F(n-2)$. All shorter words are prefixes of longer ones. The file contains $F(41)$, 267 million symbols long, with the first 34 being `abaababaabaababaababaabaababaabaab`.

**`rs.13`** We could not trace any further information on this sequence's definition, so we quote Ferragina and Navarro directly†:

> Run-Rich String Sequence $R(n)$: A measure of string complexity [...] is the number of runs. [...] It has been shown that the maximum number of runs in a string is greater than $0.944n$ and lower than $1.029n$. Franek *et al.* show a constructive and simple way to obtain strings with many runs; the $n$-th of those strings is denoted $R(n)$. The ratio of the runs of their strings to the length approaches $\frac{3}{1+\sqrt{5}} = 0.92705\ldots$.

We did not find exactly which "Franek *et al.*" paper is referred to, but whatever it is, it was published before 2010. The first 40 symbols of the sequence, space-separated for clarity, are 01001 01101 00101 00101 10100 10110 10110 10010.

**`tm29`** This is a prefix of the Thue–Morse sequence, OEIS sequence A010060. Perhaps the simplest definition of it is based on on bitwise negation: one starts with the string $t_0 =$ `0`, and repeatedly adds to this the binary inverse of the previous string, thus $t_1 =$ `01`, $t_2 =$ `0110`, $t_3 =$ `01101001`, .... The file `tm29` is $2^{28}$ bytes long, so it contains the string $t_{28}$, expressed with the characters "`a`" and "`b`". The first 32 characters are `abbabaabbaababbabaababbaabbabaab`.

* For the 5% dictionaries, simply subtract 5 percentage points. For the 1 MiB prefix dictionaries, first calculate how many percent 1 MiB≈1.049 MB is of the original file, then subtract that from the percentage given.

† http://pizzachili.dcc.uchile.cl/artificial-stats.html

Compression results are listed in Table 5.4. Unsurprisingly, all compressors performed very well on these very repetitive sequences. RLZ with the 5% dictionary performed worst overall, but if one discards the size of the dictionary (by subtracting 5 percentage points) it is on par with, and sometimes exceeds, the performance of `xz`. RLZ also performed exceptionally well with the 1 MiB prefix dictionary: the parsing itself only occupies single kilobytes of space, and the mean phrase lengths were over 710,000 for all three files. Even with the 1 MiB prefix dictionary (which is around 0.3% of the input) the total output was on par with `gzip`'s and `xz`'s output.

| File | 1 MiB prefix | 5% square | `gzip -9` | `xz -9` |
|---|---|---|---|---|
| `fib41` | 0.39% | 5.14% | 0.44% | 0.18% |
| *267.9 MB* | *3,016 B* | *370.9 kB* | *1,176 kB* | *473.3 kB* |
| `rs.13` | 0.48% | 5.19% | 0.51% | 0.16% |
| *216.7 MB* | *2,440 B* | *414.7 kB* | *1,097 kB* | *354.2 kB* |
| `tm29` | 0.39% | 5.19% | 0.53% | 0.36% |
| *268.4 MB* | *2,728 B* | *524.3 kB* | *1,420 kB* | *964.5 kB* |

**Table 5.4:** The Pizza&Chili artificial repetitive file corpus, compressed with RLZ with 1 MiB prefix and 5% square dictionaries, and also compressed with `gzip` and `xz`. Compressed RLZ file sizes do not include the size of the dictionary, but the compression factor (output÷input) does.

When it comes to compression speed, `gzip` was the fastest: all three files were compressed in about 3 seconds. `xz` took 15 seconds for `fib41`, 12 s for `rs.13`, and 25 s for `tm29`. `rlzparse` with the 1 MiB prefix dictionary took 19 seconds for `fib41`, 10 s for `rs.13`, and 13 s for `tm29`; with the 5% dictionary, it took 97 s, 90 s, and 113 s respectively, being by far the slowest.

### 5.4.2 Pseudo-real data

The Pizza&Chili corpus' "pseudo-real" data files are built by taking a 1 MiB prefix of one of the files of the non-repetitive data set (some of which we also tested compression on, in section 5.3), then repeatedly mutating it. 99 mutated segments were constructed, then they were all concatenated into a single file, with the un-mutated segment at the start.

The mutation takes the form of random character substitutions, with substitute characters taken from the same alphabet. There were three frequencies of mutation used: those files containing "`00001`" in their names had 0.001% of characters substituted, "`0001`" had 0.01%, and "`001`" had 0.1%, one in a thousand, characters substituted. There were also two approaches to mutation: those files with names ending in "`.1`" have each segment mutating the original segment (and so changes do not accu-

mulate), while those with names ending in ".2" mutated the preceding segment, and so changes accumulated throughout the file.

| File | 1 MiB prefix | 5% square | `gzip -9` | `xz -9` |
|---|---|---|---|---|
| `dblp.xml.00001.1` | 1.02% | 5.62% | 17.51% | 0.15% |
| *100 MiB* | *16.6 kB* | *658.6 kB* | *18,359 kB* | *157.2 kB* |
| `dblp.xml.00001.2` | 1.76% | 5.75% | 17.66% | 0.15% |
| *100 MiB* | *792.7 kB* | *788.6 kB* | *18,514 kB* | *156.3 kB* |
| `dblp.xml.0001.1` | 1.16% | 5.75% | 17.54% | 0.18% |
| *100 MiB* | *165,5 kB* | *789.9 kB* | *18,390 kB* | *192.8 kB* |
| `dblp.xml.0001.2` | 8.71% | 6.97% | 18.94% | 0.18% |
| *100 MiB* | *8,081 kB* | *2,071 kB* | *19,858 kB* | *184.5 kB* |
| `dna.001.1` | 2.56% | 7.30% | 27.17% | 0.51% |
| *100 MiB* | *1,637 kB* | *2.416 kB* | *28,486 kB* | *537.0 kB* |
| `english.001.2` | 62.30% | 18.69% | 42.54% | 0.55% |
| *100 MiB* | *64,273 kB* | *14,360 kB* | *44,605 kB* | *572.6 kB* |
| `proteins.001.1` | 2.58% | 7.26% | 38.50% | 0.59% |
| *100 MiB* | *1,651 kB* | *2,377 kB* | *40,369 kB* | *623.2 kB* |
| `sources.001.2` | 66.25% | 18.22% | 34.35% | 0.44% |
| *100 MiB* | *68,418 kB* | *13,870 kB* | *36,023 kB* | *464.7 kB* |

**Table 5.5:** Compressed file sizes and compression factors of the Pizza&Chili "pseudo-real" dataset, using both RLZ with 1 MiB prefix and 5% square dictionaries as well as `gzip` and `xz` at their maximum compression ratio. Compressed RLZ output sizes do not include the size of the dictionary, but the compression factor does.

The corpus that was available for download in January 2023 did not contain all available frequencies with both mutation strategies, which limits what conclusions we can draw with respect to how different mutation strategies affect RLZ's performance. Nevertheless, we can conclude the following: `xz` was by far the most efficient, `gzip` was least efficient, and RLZ did worse when mutations accumulated, but beat `gzip` in almost all cases. RLZ did particularly badly when compressing English text or source code (both human-readable and human-written data types) against the first, un-mutated document—in both `english.001.2` and `sources.001.2`, by the time the 100th segment starts, approximately 1 in 10 characters ($0.1\% \times 98$) would differ from the original. (We wonder how well it would have done if all mutations were based on the original segment.) If the size of the dictionary were discounted, RLZ was competitive against `xz` in a couple of cases.

As for runtime: `gzip` was generally fastest, taking under 5 seconds for most files, but 142 s for `dna.001.1`, 13 s for `english.001.2`, and 20 s for `sources.001.2`. `xz` took

between 20 and 25 seconds for most files, but `dna.001.1` took 60 s and `english.001.2` took 40 s. `rlzparse` with the 1 MiB prefix dictionary was fastest when its compression was the best: taking under 3 seconds with the first three `dblp` files and 8.5 with the fourth, 3.4 s with `dna.001.1`, 4.3 s with `proteins.001.1`, and about 29 s with `english.001.2` and `sources.001.2`. With the 5% dictionary, `rlzparse` took between 8 and 10 seconds with the `dblp`, `dna.001.1`, and `proteins.001.1` files, 16.5 s with `sources.001.2`, and 19 s with `english.001.2`—faster than the prefix dictionary, correlating with the superior compression factor.

### 5.4.3 Real data

The Pizza&Chili Repetitive Corpus had nine "real" files: four of them are DNA, three of them are natural-language text, and two of them are program source code.

- `cere`: 37 concatenated DNA sequences of different strains of *Saccharomyces cerevisiae*, from the Saccharomyces Genome Resequencing Project.

- `coreutils`: Source code of the GNU Core Utilities ("coreutils"): "We collected all versions 5.x of the *coreutils* package and removed all binary files, making a total of 9 versions". Manual inspection of the file has shown it to be a mix of Make files, Perl, Bourne Shell scripts (many of them automated tests), text files used as test inputs to the utilities, manual pages, and C source code. It is interesting to note that the first megabyte has no C source code in it, being just Make files and test scripts.

- `einstein.de.txt`: All versions of the German-language Wikipedia article on Albert Einstein, from creation up to the version of January 12, 2010*. The format is "wikitext", the markup of the page. The character encoding is Windows-1252, so the non-ASCII German letters *äöü* are represented with single bytes†. One newline character separates versions. We have calculated that the file contains 2130±1 revisions‡.

* The last version included is at https://de.wikipedia.org/w/index.php?title=Albert_Einstein&oldid=69172767.

† Comparing the wikitext in the file with the one found on Wikipedia, minor differences can be found caused by the character encoding conversion (Wikipedia uses Unicode), for example all en-dashes have been converted into plain ASCII hyphens. There is also a change in whitespace: paragraphs are separated with two newlines on Wikipedia, and only one in the file.

‡Counting versions on Wikipedia's version history we get 2129. Searching (with `grep`) and counting the opening words of the article we get 2131.

- `einstein.en.txt`: All versions of the English-language Wikipedia article on Albert Einstein, from creation up to the version of November 10, 2006*. Like the German version, a character set conversion to Windows-1252 has taken place, and some whitespace changes have occurred, relative to the original wikitext on Wikipedia. We have calculated that there are exactly 7500 revisions in the file.

- `Escherichia_Coli`: 23 DNA sequences of *Escherichia coli*, from the National Center for Biotechnology Information (NCBI).

- `influenza`: "78,041 [DNA] sequences of *Haemophilus influenzae*, also from the NCBI." †

- `kernel`: "We also collected all 1.0.x and 1.1.x versions of the Linux kernel, making a total of 36 versions." Mainly C source code.

- `para`: 36 DNA sequences of different strains of *Saccharomyces paradoxus*, from the Saccharomyces Genome Resequencing Project.

- `world_leaders`: A collection of *Chiefs of State and Cabinet Members of Foreign Governments* documents, which were formerly distributed as PDF files on the Central Intelligence Agency's website. These were converted to text with `pdftotext`‡. A quick and approximate count (by searching case-insensitively for "key to abbreviations") told us that the file contains at least 84 documents.

Table 5.6 shows the results of our compression tests. Like with the preceding datasets, `xz` was clearly the most effective compressor. RLZ performed very poorly with the 1 MiB prefix dictionary—more often than not, the output was bigger than

* The last version included is at https://en.wikipedia.org/w/index.php?title=Albert_Einstein&oldid=87011531.

†We have some doubts as to the accuracy of this description: the bacterium *H. influenzae* has a genome of some 1.8 million base pairs. Dividing the size of the file by that number gives about 86 sequences, while dividing by 78,041 sequences gives about 1983.7 base pairs per sequence, off by a factor of 900. Perhaps a decimal point has been mistaken for a thousands-separating comma?

‡ The conversion process was not entirely error-free, especially with graphic elements, as evidenced by the opening "words" of the first document in the file: `CENTRAL LIGEN C TEL IN IT ED STA M TES OF A ER Chiefs of State and Cabinet Members of Foreign Governments`. In the body of the documents, the text itself appears to be correct, but for the human reader formatting leaves much to be desired: the original PDFs evidently used dots to line up a government position to the corresponding name in a listing (*cf.* how some tables of contents connect a section name to a page number), and those dots have been reproduced verbatim as ASCII periods, while line terminators have not. Tables have been unhelpfully written out in column-major order. This is all to say that, although the *source* of the data is real, this collection might not be one that would be realistically constructed.

| File | Prefix | Square | Tailored | `gzip -9` | `xz -9` |
|---|---|---|---|---|---|
| `cere` | 69.21% | 17.83% | 8.90% | 26.20% | 1.14% |
| *461.3 MB* | *318 MB* | *59.2 MB* | *12.5+28.6 MB* | *120 MB* | *5.24 MB* |
| `coreutils` | 214.04% | 33.46% | 32.98% | 24.32% | 1.90% |
| *205.3 MB* | *438 MB* | *58.4 MB* | *22.8+44.9 MB* | *49.9 MB* | *3.90 MB* |
| `einstein.de.txt` | 137.01% | 5.87% | 18.91% | 31.04% | 0.11% |
| *92.8 MB* | *126 MB* | *805 kB* | *0.13+17.4 MB* | *28.8 MB* | *99.0 kB* |
| `einstein.en.txt` | 178.14% | 5.43% | 29.18% | 35.00% | 0.07% |
| *467.6 MB* | *832 MB* | *2.03 MB* | *0.15+136 MB* | *163 MB* | *323 kB* |
| `Escherichia_Coli` | 68.69% | 45.99% | 36.39% | 27.98% | 4.67% |
| *112.7 MB* | *76.4 MB* | *46.2 MB* | *4.9+36.1 MB* | *31.5 MB* | *5.26 MB* |
| `influenza` | 45.09% | 13.00% | 123.40% | 6.87% | 1.34% |
| *154.8 MB* | *68.8 MB* | *12.4 MB* | *4 kB+191 MB* | *10.6 MB* | *2.07 MB* |
| `kernel` | 104.71% | 18.42% | 42.96% | 26.90% | 0.81% |
| *258.0 MB* | *269 MB* | *34.6 MB* | *7.2+104 MB* | *69.3 MB* | *2.09 MB* |
| `para` | 71.94% | 23.12% | 31.91% | 27.04% | 1.46% |
| *429.3 MB* | *307 MB* | *77.8 MB* | *11.9+125 MB* | *116 MB* | *6.26 MB* |
| `world_leaders` | 40.11% | 17.53% | 26.18% | 17.65% | 1.29% |
| *47.0 MB* | *17.8 MB* | *5.89 MB* | *3.7+8.6 MB* | *8.29 MB* | *607 kB* |

**Table 5.6:** Compressed file sizes and compression factors of the Pizza&Chili real dataset, using both RLZ with 1 MiB prefix, 5% square, and input-specific tailored dictionaries as well as `gzip` and `xz` at their maximum compression ratio. The compression ratios include the size of the dictionary in the calculation. The output sizes for the "prefix" and "square" columns do not include the size of the dictionary (as it's either constant or easily derived from input size). In the "tailored" column, the size of the dictionary is expressed before the plus sign and the size of the RLZ output file is after it.

the input, and for the `coreutils` file, the output was over *double* the size of the input. (After inspecting the file itself, this is not surprising: the first mebibyte is mainly just Make files and shell scripts, completely lacking C source code, and so attempts to compress C end up with very short phrases—Make files do not include strings such as "`#ifndef`".)

RLZ fared better with a more varied 5% dictionary, beating `gz` in most cases. The largest improvements were seen in the `einstein` Wikipedia version history datasets: if the size of the dictionary itself were discounted, RLZ achieved a compression ratio under 1%.

We considered the possibility that neither the 1 MiB prefix nor the 5% square dictionary are particularly useful, or particularly realistic, dictionaries to compress with. Recall the formulation of the files: they are concatenations of 9 to 78,041 similar

documents. It is therefore reasonable to presume that one, or a small set, of these documents would work well as a dictionary for the others: identical sections would be parsed into long phrases, and much of the factorized output would involve coding the differences to the dictionary. We thus created *tailored* dictionaries for each of these documents, which simulate using one or a couple of the documents as a dictionary. We calculated the "average document length", by dividing the size of the file by the number of documents. For the DNA and code files (*i.e.* all except the `einstein`s and `world_leaders`) we took a prefix of that average document length (arbitrarily rounded up to a multiple of 1024) and used that as a dictionary. For the `influenza` file, we also took a *suffix* of the average document length, as the dictionary was so small otherwise. For the natural-language files, we produced dictionaries: with `world_leaders`, we took the first and the last document, and with the `einstein`s, we took the first version, the last version, and the first version that started after the midpoint. The "tailored" column in Table 5.6 displays the results of compressing with these tailored dictionaries; the size of the dictionary is given along with the size of the compressed file. We note that the tailored dictionary is larger than the 5% dictionary for all files except `coreutils` and `world_leaders`.

For the DNA inputs of `cere` and `Escherichia_Coli` and the `coreutils` data set, the tailored dictionary worked better than the others, and the output size was on par with `gzip`'s. The `para` DNA input, the `world_leaders` text input, and the `kernel` code input compressed better than with the prefix but worse than with the 5% square dictionary. The square dictionary was also best for the `einstein` files: we presume that with more frequent sampling (say, including every 500th or even 100th version, rather than 3 versions out of 2130 or 7500) the output size will decrease closer to the levels of the 5% dictionary case. (Note, however, how even with such a small dictionary—150 kilobytes—a compression ratio better than `gzip`'s was achieved.) The tailored dictionary, 4 KiB in size, performed very poorly with the `influenza` input—with a prefix dictionary that was 1/78th the size of the file we got a compression ratio of 34.41%, sitting between the 1 MiB prefix and the 5% square dictionaries.

As for compression times, the general trend was that RLZ was faster when compressing DNA (`cere`, `Escherichia_Coli`, `influenza`, `para`), and the more-traditional compressors were faster when compressing text (whether English, German, or source code). With text, `gzip` was always the fastest, with a median compression speed of 12 MB/s. `xz` took about twice to thrice as long, with a median speed of 4 MB/s. RLZ with the 5% dictionary was roughly 30% faster than `xz`, and RLZ with the 1 MiB dictionary lagged behind, being between 4 and 8 times slower than `gzip`. With DNA, RLZ with a 5% dictionary was fastest: 177 seconds for `cere`, 200 s for `para`, 36 s for `influenza`; `Escherichia_Coli` took 56 s with the 1 MiB and 75 s with the 5%

dictionary. `xz` took 1.2 to 2.7 times as long as RLZ-5% did, and `gzip` took roughly 1.2 times as long as `xz`. RLZ with the tailored dictionary was typically faster than RLZ-5% by a couple of seconds, correlating with the slightly smaller dictionary size; the major exceptions were `kernel` and `para`, which were slower by 50 and 60 seconds, respectively.

### 5.4.4 Log files

The Pizza&Chili log file collection included nine log files, divided into two groups: execution traces (three files, all larger than 100 MiB) and program logs (six files). The program logs are from the Loghub dataset*, and have been limited to a prefix 100 MiB in length. The files `Horspool`, `NQueens`, and `Quicksort` are the execution traces: they list the order that functions were entered and exited during the execution of a pattern matching algorithm, a solver of the $n$ queens problem, and a sorting algorithm sorting random numbers, respectively. `BGL` is from the Blue Gene/L supercomputer, `Thunderbird` is from the Thunderbird supercomputer, `Windows` and `Android` are system logs from some installation of their respective operating systems, and `HDFS_1` is a log from the Hadoop distributed file system. `EDGAR` is a 100 MiB prefix of the EDGAR log file dataset, published by the US Securities and Exchange Commission, pertaining to search traffic in a database.

The execution traces are far more compressible than the system logs: each line of them contains one of a small number (12 to 16) of fixed messages, and half of them are identical to the other half by all except their first character. The system logs' lines tend to have unique features such as timestamps, session numbers and randomly-chosen identifiers, so they are less amenable to compression in general and RLZ parsing in particular.

Like with the other datasets, we compressed the files with both a 1 MiB prefix dictionary and a 5% square dictionary†, and compare the results with `gzip`'s and `xz`'s performance. We used the fixed-size `32x2` format, which results in 8 bytes per token. The results of our tests are shown in Table 5.7.

The three execution traces clearly compressed much better than the system logs, especially with the 1 MiB prefix dictionaries: the dictionaries were very likely to contain most of the 12 to 16 fixed phrases. It is interesting to once again note that, if the size of the dictionary were ignored, `rlzparse`'s output was smaller than even `xz`'s output was, for these three files.

*https://github.com/logpai/loghub

†For the 100 MiB dictionaries, the 5% square dictionaries had $n = l = 2289 \approx \sqrt{5 \times 2^{20}}$. The largest file, 572 MB in size, had $n = l = 5347$.

| File | 1 MiB prefix | 5% square | `gzip -9` | `xz -9` |
|---|---|---|---|---|
| `Android` | 110.82% | 29.88% | 13.15% | 4.82% |
| *104.9 MB* | *115.1 MB* | *26.1 MB* | *13.8 MB* | *5.06 MB* |
| `BGL` | 147.70% | 36.12% | 7.34% | 4.33% |
| *104.9 MB* | *153.8 MB* | *32.6 MB* | *7.70 MB* | *4.54 MB* |
| `EDGAR` | 63.17% | 37.27% | 12.54% | 7.53% |
| *104.9 MB* | *65.2 MB* | *33.8 MB* | *13.2 MB* | *7.89 MB* |
| `HDFS_1` | 68.17% | 36.55% | 9.61% | 6.42% |
| *104.9 MB* | *70.4 MB* | *33.1 MB* | *10.1 MB* | *6.73 MB* |
| `Horspool` | 0.41% | 5.20% | 0.50% | 0.24% |
| *571.9 MB* | *1.30 MB* | *1.13 MB* | *2.85 MB* | *1.39 MB* |
| `NQueens` | 0.84% | 5.27% | 0.52% | 0.36% |
| *302.6 MB* | *1.50 MB* | *831 kB* | *1.56 MB* | *1.09 MB* |
| `Quicksort` | 2.75% | 5.27% | 0.56% | 0.42% |
| *381.5 MB* | *9.45 MB* | *1.01 MB* | *2.13 MB* | *1.59 MB* |
| `Thunderbird` | 75.51% | 32.42% | 9.04% | 4.27% |
| *104.9 MB* | *78.2 MB* | *28.8 MB* | *9.48 MB* | *4.47 MB* |
| `Windows` | 176.90% | 8.84% | 5.53% | 0.51% |
| *104.9 MB* | *184.4 MB* | *4.03 MB* | *5.79 MB* | *534 kB* |

**Table 5.7:** Compressed file sizes and compression factors of the Pizza&Chili log dataset, using both RLZ with 1 MiB prefix and 5% square dictionaries as well as `gzip` and `xz` at their maximum compression ratio. Compressed RLZ output sizes do not include the size of the dictionary, but the compression ratio does.

The prefix dictionaries performed very poorly with the system logs. This can be explained by the fact that similar messages in system logs tend to be clustered: if a short-duration event happens, then a cluster of messages relating to that event will be logged for the duration, after which the messages will not be seen again. Therefore, as a prefix only contains a subset of possible messages, it performs poorly, and a more frequently sampled dictionary performs better.

As for runtime: the 100 MiB dictionaries were parsed by `rlzparse` in a mean time of 41 seconds with the 1 MiB prefix dictionary (with a minimum time of 31.9 s and maximum time of 52.8 s) and 34.5 seconds with the 5% square dictionary (with a minimum of 13.2 s and maximum of 45.3 s); the average rate of compression was 2.5 MB of input per second with the prefix and 3.0 MB/s with the square dictionary. For comparison, `gzip` compressed at 17.6 MB/s and `xz` at 2.6 MB/s. The stack traces were compressed faster: `gzip` at a mean rate of 130 MB/s and `xz` at 24 MB/s, while `rlzparse` processed them at 4.9 MB/s with a prefix dictionary and 2.4 MB/s with a square dictionary.

## 5.5 Influence of output format

So far, all of our testing has used `rlzparse`'s "`32x2`" output format, where each token is represented by a pair of two fixed-size 32-bit unsigned integers. This has the advantage that the number of phrases is easily calculated from the size of the file, by dividing the size by eight. The disadvantage is inefficiency: literal phrases, very short phrases, and references to early locations in the dictionary lead to much of the output bits being zero. Since mean phrase lengths have, for most inputs, been rather small (under 100 for 67% of the runs of `rlzparse` on the entirety of the Pizza&Chili repetitive corpus with four different dictionary sizes (1 MiB prefix, 0.1%, 1%, 5% squares), and under 20 for 42% of them), this waste of space adds up—with a variable-byte coding where numbers below 128 would be expressed with a single byte, about 3/8ths of the used space could be saved on for most files, and thus compression ratio would be greatly improved.

To test the effect of choosing `vbyte` over `32x2` on performance, we took a selection of 7 previously-compressed files and re-compressed them with the `--format vbyte` flag. Once again, we use two dictionaries: a 1 MiB prefix dictionary and a 5% square dictionary. We measured both the size of the output and the runtime of the program, as well as `rlzunparse`'s decompression time.

Table 5.8 shows the size of the compressed files, the mean phrase lengths of the parsings, the compression ratio (including dictionary size) of the compressed files, and the size of the `vbyte`-format file as a percentage of the `32x2`-format file. With this set of files, a 40% to 50% saving in space was recorded: the lower the mean phrase length (and thus the worse the compression ratio), the smaller the `vbyte` file was relative to the `32x2` one.

Table 5.9 shows the compression and decompression times of the same files. In 10 compression runs out of 14, compressing with `vbyte` encoding was marginally faster than with the `32x2` format; time savings were on the order of 1 to 2%, a fraction of a second in most cases. In the remaining 4 cases, compression was slower by between 0.2 and 1.25%, or 0.02 to 1.19 seconds slower. (These 4 cases all involved the three bioinformatic files: `proteins.001.1`, `Escherichia_Coli` and `para`. Compression ratio, mean phrase length, or file size does not appear to be a common factor.)

Conversely, in all 14 cases, decompressing the `vbyte`-encoded file was *slower*, albeit again the effect is small, under 0.1 seconds in 9 cases and 1.26 seconds in the slowest case. (Proportionally the slowdown of compression time is quite large: between 1.3% and 12.6% slower.)

| Input file | $\lvert S\rvert_{\text{MB}}$ | $\lvert D\rvert$ | MPL | $\text{Size}_{32}$ | $\text{Size}_{\text{vb}}$ | $\frac{\text{vb}}{32}\%$ | $\text{CR}_{32}$ | $\text{CR}_{\text{vb}}$ |
|---|---|---|---|---|---|---|---|---|
| `english.001.2` | 104.9 | $2^{20}$ | 13.1 | 64.27 | 32.03 | 49.8 | 62.3 | 31.6 |
| `proteins.001.1` | 104.9 | $2^{20}$ | 507.8 | 1.65 | 0.91 | 55.3 | 2.6 | 1.9 |
| `sources.001.2` | 104.9 | $2^{20}$ | 12.3 | 68.42 | 34.02 | 49.7 | 66.2 | 33.4 |
| `coreutils` | 205.3 | $2^{20}$ | 3.8 | 438.33 | 211.80 | 48.3 | 214.0 | 103.7 |
| `Escherichia_Coli` | 112.7 | $2^{20}$ | 11.8 | 76.35 | 38.05 | 49.8 | 68.7 | 34.7 |
| `einstein.en.txt` | 467.6 | $2^{20}$ | 4.5 | 831.96 | 412.42 | 49.6 | 178.1 | 88.4 |
| `para` | 429.3 | $2^{20}$ | 11.2 | 307.79 | 153.40 | 49.8 | 71.9 | 36.0 |
| `english.001.2` | 104.9 | $2289^2$ | 58.4 | 14.36 | 8.51 | 59.3 | 18.7 | 13.1 |
| `proteins.001.1` | 104.9 | $2289^2$ | 352.9 | 2.38 | 1.50 | 63.0 | 7.3 | 6.4 |
| `sources.001.2` | 104.9 | $2289^2$ | 60.5 | 13.87 | 8.26 | 59.6 | 18.2 | 12.9 |
| `coreutils` | 205.3 | $3203^2$ | 28.1 | 58.44 | 35.15 | 60.1 | 33.5 | 22.1 |
| `Escherichia_Coli` | 112.7 | $2373^2$ | 19.5 | 46.20 | 26.79 | 58.0 | 46.0 | 28.8 |
| `einstein.en.txt` | 467.6 | $4835^2$ | 1838.7 | 2.03 | 1.36 | 66.9 | 5.4 | 5.3 |
| `para` | 429.3 | $4632^2$ | 44.1 | 77.80 | 47.88 | 61.5 | 23.1 | 16.2 |

**Table 5.8:** Output sizes of a selection of Pizza&Chili Repetitive Corpus files when compressed into the `32x2` and `vbyte` formats. Column legend: $\lvert S\rvert$ is the input file's size (in megabytes), $\lvert D\rvert$ is the dictionary's size (in bytes), "MPL" is the mean phrase length, "Size" is the output's size (in megabytes), "$\frac{\text{vb}}{32}\%$" is the size of the `vbyte` output divided by the `32x2` output (expressed as a percentage), and "CR" is the compression ratio (output+dictionary/input). The subscripts "32" and "vb" refer to `32x2` and `vbyte`. The first section of files, those with a dictionary size of $2^{20}$ bytes (1 MiB), use a prefix dictionary. The second section of files use a 5% square dictionary.

| | 1 MiB prefix dictionary | | | | 5% square dictionary | | | |
|---|---|---|---|---|---|---|---|---|
| Input file | $\text{ct}_{32}$ | $\text{ct}_{\text{vb}}$ | $\text{dt}_{32}$ | $\text{dt}_{\text{vb}}$ | $\text{ct}_{32}$ | $\text{ct}_{\text{vb}}$ | $\text{dt}_{32}$ | $\text{dt}_{\text{vb}}$ |
| `english.001.2` | 29.90 | 29.54 | 1.65 | 1.70 | 19.20 | 19.12 | 1.34 | 1.41 |
| `proteins.001.1` | 4.20 | 4.14 | 1.23 | 1.27 | 9.78 | 9.80 | 1.20 | 1.25 |
| `sources.001.2` | 29.28 | 28.65 | 1.63 | 1.72 | 16.73 | 16.49 | 1.35 | 1.39 |
| `coreutils` | 116.57 | 114.80 | 4.95 | 5.44 | 65.56 | 65.43 | 2.83 | 3.06 |
| `Escherichia_Coli` | 58.28 | 57.59 | 1.82 | 1.92 | 74.98 | 75.92 | 2.11 | 2.19 |
| `einstein.en.txt` | 316.26 | 310.50 | 10.05 | 11.31 | 58.37 | 55.64 | 5.35 | 5.42 |
| `para` | 227.72 | 228.91 | 7.16 | 7.45 | 201.21 | 202.27 | 6.25 | 6.68 |

**Table 5.9:** Compression and decompression runtimes of a selection of Pizza&Chili Repetitive Corpus files when compressed into the `32x2` and `vbyte` formats. Legend: "ct" stands for compression time, "dt" for decompression time, subscript "32" for the `32x2` format, and subscript "vb" for the `vbyte` format. All units are in seconds.

Our explanation for why compression with `vbyte` is faster, but decompression is slower, is that the time savings obtained by having less data to write makes up for the overhead of the variable-byte encoding: writing is just so slow. On the other hand, reading from storage is comparatively quick, and all `rlzunparse` has to do is decode the encoded number pairs and write a sequential block of data. The blocks of data are the same in both cases, so the slowdown is in the decoding. `32x2` decoding is done 8 bytes at a time, with a `reinterpret_cast` from a length-8 array of bytes to a length-2 array of 32-bit integers, so in fact no arithmetic is done: the typecast is for the benefit of the compiler. Conversely, `vbyte` decoding requires a minimum of one Boolean operation (a bitwise-AND with 128), one equality test, and one addition per byte; most bytes cause a second addition, a second bitwise-AND (with 127), another comparison, and one bitwise shift.

To summarize, compared with `32x2`, compressing with the `vbyte` format brings large savings (40 to 60%) in disk usage, a marginal improvement ($\approx 1\%$) to compression speed, and a minor worsening ($\approx 5\%$) of decompression speed. Considering the space savings, we suggest that using `vbyte` is generally worth the time tradeoff—however, fixed-width encoding is not without its advantages, and the choice of output format will vary by application. `32x2` remains `rlzparse`'s default.

## 5.6 Dictionary construction parameters

So far, our testing has used either prefix dictionaries or square dictionaries, and we also settled on one-mebibyte prefixes and 5%-of-input squares. A natural question to ask is: could we do better with different dictionary shapes? What about different sizes—is 5% overkill, or not large enough? Of course, it is unlikely that one set of parameters works for all inputs, but maybe we could derive some rules of thumb?

As a reminder from Section 3.2, a randomly-sampled rectangular dictionary is built from two parameters. For brevity we these call $n$ and $l$: the number of samples in the dictionary, and the length of each sample. However, a different but equivalent formulation would have the parameters $|D|$ and $n/l$: the size of the dictionary and its aspect ratio. With random sampling there is also the element of luck: different invocations of `builddict` with the same parameters produce a different dictionary, and it's entirely possible for one dictionary to do far better (or far worse) at representing the input than another. In the interest of time we have decided to ignore the luck element, and we simply assume that the first dictionary produced by `builddict` is representative of an "average" dictionary—as it turns out, a couple of outliers in this section's plots are probably due to particularly (un)lucky dictionaries, rather than stumbling upon magic parameters.

Therefore, our search space is two-dimensional: for a given input file, find out what size and aspect ratio the best-performing dictionary uses. (By "performance" we of course mean compression performance: how small the input is relative to the output.) *Also* in the interest of time*, we have not searched the entirety of the space, but merely two lines through it. In the first, we fixed the dictionary size parameter to 4% of the input (or exactly 8 MiB, as all the input files were 200 MiB), and varied the aspect ratio. In the second, we did the opposite: we picked a fixed aspect ratio ($n$:$l$ = 200:1) and varied the dictionary size.

We used six files as input, all from the Pizza&Chili corpus [4]. Three of them are from the regular corpus: `pc-dna`, `pc-english`, and `pc-sources`. The other three are from the repetitive corpus' "real" collection: `cere`, `einstein`(`.en.txt`), and `kernel`. (Additionally, for the dictionary size test, we used the `Windows` file from the repetitive corpus' log file collection—we did not use it for the aspect ratio compression tests because it was too small.) We thus have three types of file from both corpora: DNA, natural-language (English, encoded with an 8-byte character set), and source code (mostly C). We have color-coded these in Figures 5.4–5.9: DNA is blue, English is red, and source code is green.

### 5.6.1 Dictionary size

We have discretized the size parameter into twelve points, in a 1–2–5–10 pattern ranging from 0.005% to 20% of the input size. We used whole input files, with sizes ranging from 104.9 MB (`windows`) to 2.21 GB (`einstein`). We used a fixed aspect ratio of 200:1, meaning the dictionary had 200 times more samples than the sample length was.

Because the $n$ and $l$ parameters, and the dictionary size, must be integers, the target sizes from 0.005% to 20% are merely approximations. The exact calculations were $l = \lfloor\sqrt{|S|/200}\rfloor$, $n = \lceil|S|/l\rceil$. With these calculations, the actual dictionary size would never be smaller than the target size, nor would the aspect ratio be under 200:1—$n$ would always be just a bit too large. However, we consider this error to be negligible: the largest relative error occurred in the 0.005% case, where a dictionary was 0.0538% too big (being 6 bytes larger than the target of 10,543 bytes); the largest aspect ratio error was also in the 0.005% case (at 234:1 instead of 200:1, since that dictionary had $n$ = 2339 and $l$ = 10 rather than $n$ = 2162.47 and $l$ = 10.81); and the largest absolute error was in the 20% case, where a 442-megabyte dictionary of $n$ =297,497, $l$ = 1486 was 1431 bytes too long.

*Our Lenovo ThinkPad T460 with an i5-6200U processor took about 11 hours to compute the results plotted in Figure 5.4 and 12.5 hours to compute those in Figure 5.7.

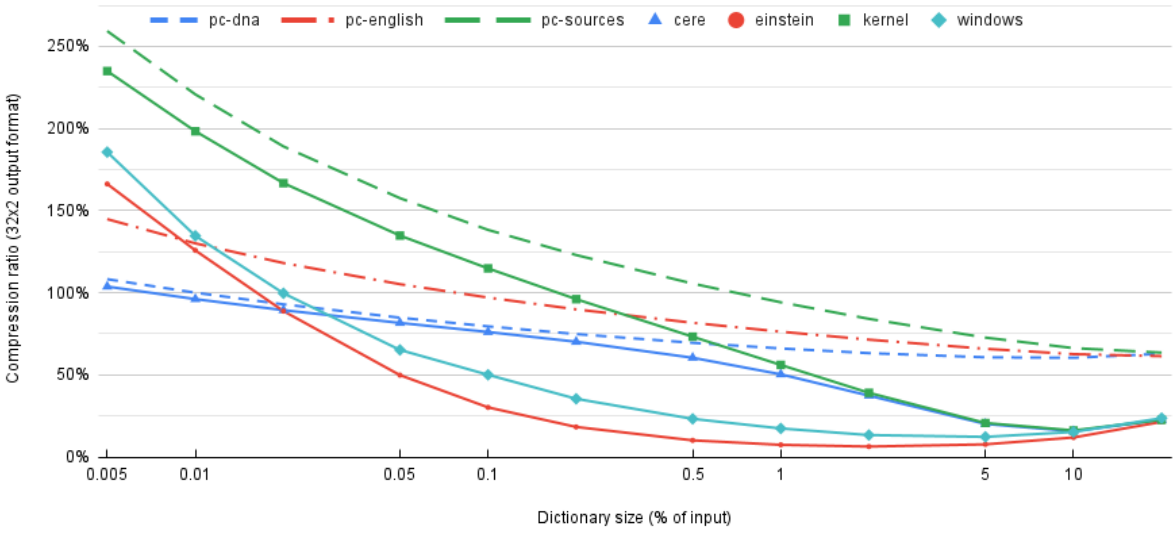


**Figure 5.4:** Compression ratio as it varies with dictionary size. Input file size varies from 100 MiB to 2.2 GB. Dictionaries are random and rectangular, with an $n$:$l$ ratio of 200:1. Output format was `32x2`. Compression ratio includes the size of the dictionary. The minimum is at 2% on the `einstein` line, with a value of 6.33%.

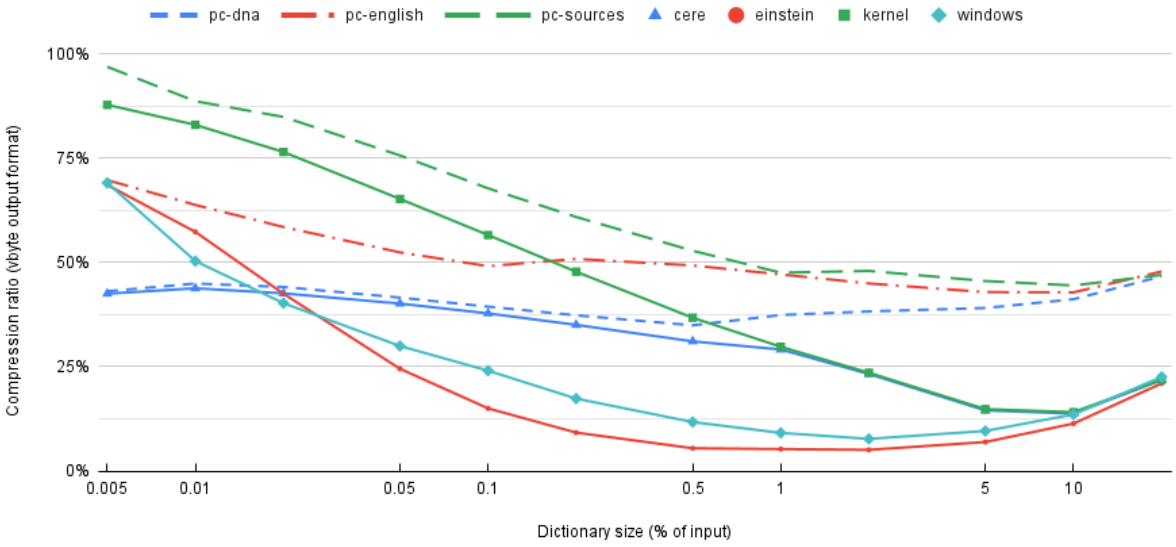


**Figure 5.5:** Plotting compression ratio as it varies with dictionary size, with the same inputs and dictionaries as in Figure 5.4, but with the `vbyte` output format. The minimum is on the `einstein` line, at 2%, with a value of 5.06%.

The results in terms of compression ratio are plotted in Figures 5.4 and 5.5; the former uses the default `32x2` output format, the latter uses `vbyte`. Both plots were derived from the same set of data: `rlzparse` was instructed to produce `vbyte` output, and then a small utility was used to count the number of phrases in the output, from which was calculated the size in the `32x2` format.

As a group, the three files from the regular, non-repetitive corpus compressed worse than the repetitive corpus files, and they ended up all converging to a compression ratio of about 63% (with the `32x2` format) or 47% (with the `vbyte` format). The Wikipedia version history collection, `einstein`, was most easily compressed: it achieved a 6.33% compression ratio, including a 2% dictionary. After that point, dictionary size increases were unproductive, as the increase in dictionary size undoes the decrease in the number of RLZ phrases.

The plots we see agree with the theoretical analysis of reference construction in [5], with the compression ratio falling sharply as dictionary size increases, then rising slowly after the minimum. (Keep in mind that the plots have a logarithmic scale. With a linear scale, the lines of Figures 5.4 and 5.5 have the shape of a rounded L.) We can conclude that it is better for a dictionary to be slightly too large than too small.

Like discussed in Section 5.5, the `vbyte` output format is much more efficient at encoding the RLZ phrases than the `32x2` default. Observe how *all* of the inputs are incompressible with the first few dictionary cases with the `32x2` format, and it is not until the 1% dictionary that all are compressed, while with the `vbyte` format, all dictionaries compress the output at least a little. Also note the bumps in the `pc-dna` and `pc-english` files at 0.1% and 0.5% in Figure 5.5: the compression ratio achieves a local minimum, before rising again. These are the points where the output starts to have a considerable amount of phrases with a length greater than 127, thus necessitating a two-byte length field when encoded.

Figure 5.6 plots the speed that the inputs were compressed at—higher points are better. We plot speed instead of actual runtime because of the variation in input size: the slowest `pc-english` instances took over an hour to compress all 2.2 GB, while the fastest `windows` instances took only 25 seconds to compress 104.9 MB. Slower speed correlates with a worse compression ratio; the compressor loop spends more time in the early stages of token search, searching the entire width of the dictionary for the initial matching symbols.

Slower speed also correlates with a larger dictionary, but `cere` and `kernel` are impressive counterexamples, with their fastest speeds obtained at the largest dictionaries. We suspect this is because at those sizes, the dictionary cannot help but contain most of a complete version of the documents that the files are built up from (i.e. a complete genome, or a complete version of the Linux kernel—albeit a kind of “hybrid”,

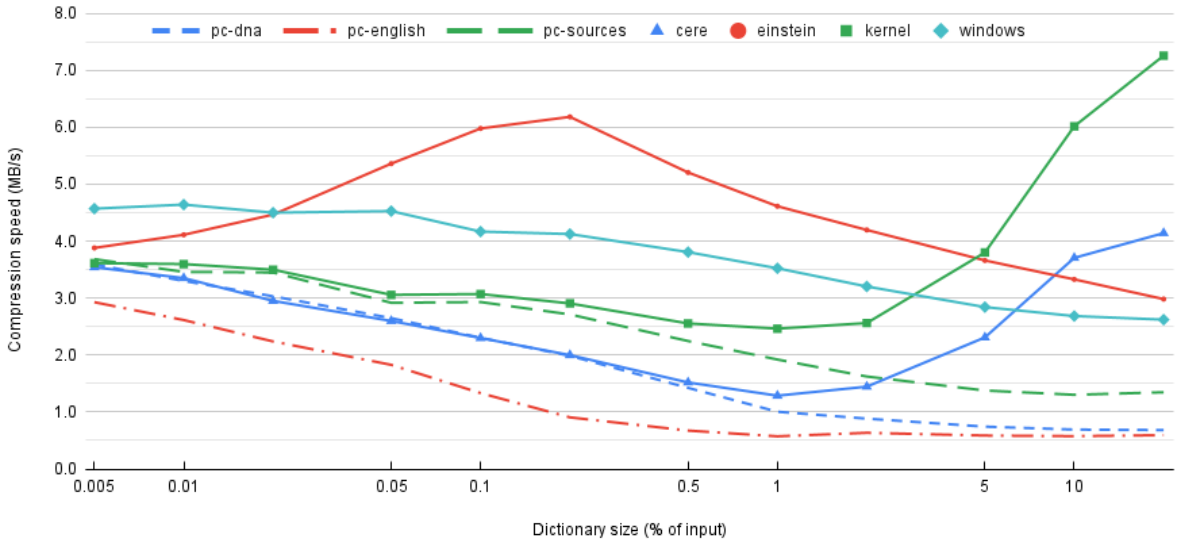


**Figure 5.6:** Same input as in Figures 5.4 and 5.5, but measuring compression speed, in megabytes of input per second. The maximum is on the `kernel` line, at 20%, with a value of 7.25 MB/s. The `vbyte` output format was used during these timing benchmarks.

consisting of parts of many different versions). Additionally, unlike the `einstein` file (which slows down after the 0.2% dictionary point), they do not yet contain excessive repetition at those sizes, and thus the width of the search range narrows quickly—conversely, the versions of Einstein's Wikipedia article that are included are between 40 and 70 kilobytes each, and they change slowly, so the dictionary may contain dozens of identical versions of the same strings.

### 5.6.2 Aspect ratio

For aspect ratio calculations we used a set of six files, leaving out the `windows` log file. To produce accurate time measurements, and to keep compression times reasonable, we truncated all six inputs to a 200 MiB (209.7 MB) prefix and compressed that. (The asterisks in the legend of Figures 5.7 to 5.9 refer to these truncated prefixes.)

We settled on a 4% dictionary, which comes out to a size of 8 MiB. We chose 4% instead of a rounder number like 5% because 4% gave us a dictionary with a power of two length—$2^{23}$ specifically—and thus easier to divide up with different aspect ratios with the scheme we used. Previous experiments on a smaller, 1%=2 MiB dictionary, produced barely-compressed input with the three non-repetitive inputs, so we wanted something a bit larger, and the results of Figure 5.4 indicate that around 5% is a reasonably good dictionary size for all of our inputs, without being too large (and thus slow).

We tested with 44 different aspect ratios. Starting from an extreme of one sample of length $2^{23}$, we would double the aspect ratio, by multiplying the number of samples by $\sqrt{2}$ and dividing the length by the same amount. Thus, every other (odd-numbered)

parameter set would be a neat pair of powers of two ((1, $2^{23}$), then (2, $2^{22}$), then (4, $2^{21}$), and so on), while the intermediate (even-numbered) parameters would be irrational (($\sqrt{2}$, $2^{22.5}$), ($2\sqrt{2}$, $2^{21.5}$), etc.). Were we to only include the exact powers of two (doubling $n$ while halving $l$ for each step), we would have skipped over some interesting aspect ratios (most notably the square 1:1 dictionary, which has $n = l = 2^{11.5} \approx 2896.3$). In retrospect this was unnecessary, as dictionaries with aspect ratios ranging from a moderately wide 1:64 to a moderately tall 64:1 behaved much the same way, and only at the extremes was performance notably different. We stopped the aspect ratio parameter sequence at $2^{21}$:1, instead of going all the way to $2^{23}$:1, for at this point the length of each sample was only 2, and compression was negative for all except DNA inputs. Additionally, the bulk of testing runtime was spent on dictionary generation at this point, as `builddict`'s random algorithm (Algorithm 5 in Section 3.1) passed the threshold from almost-immediate termination to indefinitely looping. We had to resort to a quickly-written deterministic routine for the narrowest dictionaries, that simply divided the entirety of the input into a couple of million segments and picked the first two (or three, or four) bytes from each of them.

We dealt with irrational parameters by rounding them up (e.g. for the square case, $n = l = \lceil 2^{11.5} \rceil = 2897$), generating an overlong dictionary, then taking a prefix of length $2^{23}$ of that dictionary. It was important to have dictionaries be the same size, as previous experiments with variably-sized dictionaries (where we rounded down instead of up) performed markedly differently, with noticeably faster runtimes and noticeably lower compression ratios—these showed up as zig-zags in the plots. This was especially evident at the extremes, where $n = \lfloor \sqrt{2} \rfloor = 1$ produced a dictionary that was 30% smaller than the target size.

Figure 5.7 shows the compression ratio of all six files, with the `32x2` output format, with all 44 dictionary parameter sets. Figure 5.8 has the same files but uses the `vbyte` output format, and lacks the last (2M, 1) parameter set.

Again, the three non-repetitive files compressed worse than the repetitive files. We were surprised by the flatness of the curves: for most of the inputs, it did not matter whether the dictionary was tall or wide, and 1:64 did as well as 64:1 did. Only at the extremes did differences manifest: with only one or a couple of long samples in the dictionary, natural-language text compressed poorer than with more-square dictionaries, but then the DNA collection `cere` and code collection `kernel` much preferred a single long sample, achieving some of their best compression ratios at the left edge. We suspect this is because the target size of 8 MiB is close to the size of one version of their corresponding documents—`kernel` is 258 MB and has 36 versions in it (≈7.1 MB each), and `cere` is 461 MB and has 37 versions (≈12.5 MB each). More frequent and shorter sampling may (and in all likelihood does) include segments

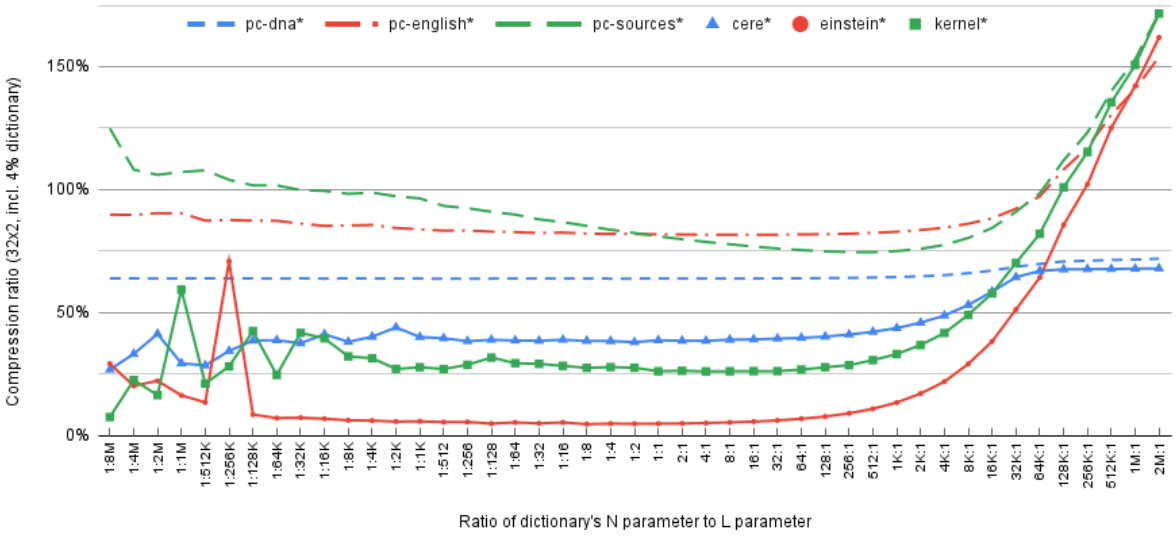


**Figure 5.7:** Compression ratio as it varies with dictionary aspect ratio. Dictionary size is constant, at 4%. Input size is also constant, at 200 MiB. Output format is `32x2`, and compression ratio includes dictionary size. Aspect ratios are expressed as $n{:}l$, number first, length second. The abbreviations "K" and "M" are binary, $2^{10}$ and $2^{20}$: "1M" means 1,048,576.

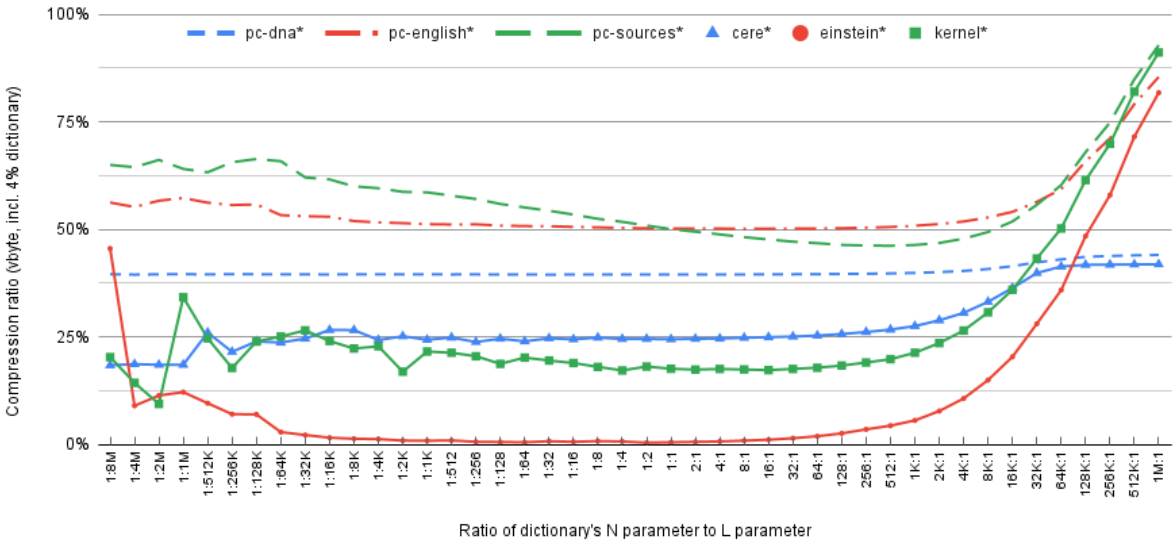


**Figure 5.8:** Same files as Figure 5.7, but with the `vbyte` output format. Dictionaries have been generated anew.

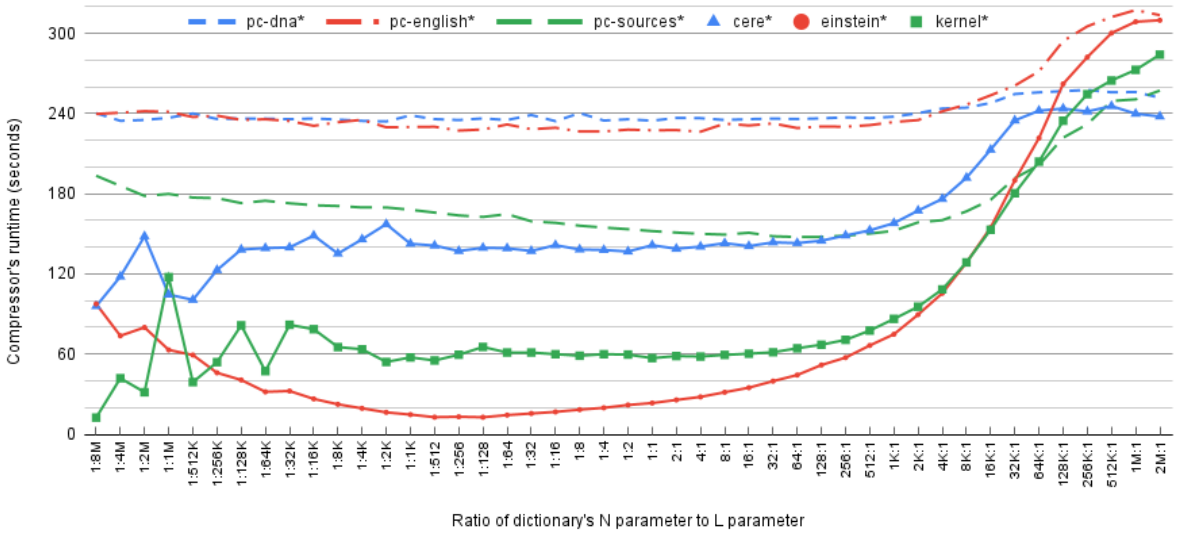


**Figure 5.9:** Time it took `rlzparse` to compress the files in Figure 5.7, in seconds. The fastest time of 12.4 seconds corresponds to a speed of 16.9 MB of input per second, and the slowest time of 317.6 seconds corresponds to a speed of 660 kB/s.

of identical text from different versions, thus in effect throwing away some dictionary capacity in redundancies.

We expected to see more of the input files to show a U-like shape in their plots, with very wide and very tall dictionaries both being inefficient, then gradually sloping down to a minimum before turning back up again as sample length falls. We were surprised at how one-sided these plots were; only the `pc-sources` and `einstein` files had a shape akin to what we expected, with the rest being quite flat after some initial instability at the start, until their predicted sharp rise to incompressibility with very short samples. The DNA file `pc-dna`, which is a single genome rather than a collection of versions, seemed indifferent to what dictionary it was compressed with: it had a compression ratio of just under 64% up to a ratio of 256:1, after which it worsened very slightly to 72%. The `cere` collection is also remarkably flat, lacking the steep rise to incompressibility. We suspect this flatness is an intrinsic property of DNA data, where the alphabet is small and the distribution of words is flat (i.e. no particular $n$-gram is any more unlikely than another: the concatenation of short DNA samples is likely also a useful DNA sample).

Because of an oversight in our benchmarking script, we had to run this suite of tests twice, once for each output format. Thus, Figures 5.7 and 5.8 are based on different dictionaries, and thus while the approximate curve shape is the same, they are not exactly comparable (unlike Figures 5.4 and 5.5, which *do* use the same dictionaries). The latter plot also lacks the 2M:1 aspect ratio case. This is why we are confident in our assessment that the spike at 1:256K in the `einstein` line of Figure 5.7 is due

to an unlucky, poorly-representative dictionary: a corresponding spike is missing in Figure 5.8, and the change in coding method is insufficient to hide the spike*.

### 5.6.3 Closing remarks on dictionary parameters

We have tested both varying dictionary sizes and varying aspect ratios on two sets of three types of input files. We have not searched the entire 12-by-44 two-dimensional parameter space that we defined, but rather two perpendicular lines through it—and while we consider our results informative, the picture we have is limited. Nevertheless, we are confident enough with our results that we can draw up conclusions and rules of thumb for randomly-sampled rectangular dictionary construction.

Our concise suggestions for dictionary parameters are this: pick a dictionary size somewhere around 5 percent of the input file size, preferably too large than too small. Divide it up so that the dictionary is reasonably tall—square dictionaries are an adequate starting point, as would having an aspect ratio between 50:1 to 500:1. Strictly structured but varying input data (e.g. source code) seems to benefit from more frequent sampling, but do not let sample length become too short—a length of 1000 is good, 100 may still work, but 10 will produce poor results.

There is much room for nuance, though, and picking good parameters requires understanding of the data to be compressed. We consider three main factors: whether the input is a collection of versions, if it is then how big are those versions, and how structured the text of the input is.

If the input is *not* a collection of similar versions, and it is not known to be highly repetitive (like the artificial mathematical binary sequences), then options are more limited. Bigger dictionaries of around 10% are probably useful here. Files that are collections of versions of constituent documents can get away with smaller dictionaries. At a minimum, the dictionary should be a couple of times larger than an average constituent document, and the more documents there are in the input file, the smaller the dictionary can be relative to the size of the whole collection.

What we mean by "structure of the text" is how amenable the input is to random sampling: is it likely or unlikely that the substring surrounding the concatenation point of two random samples occurs in the input text? That is: for samples $a_1 \ldots a_i a_j a_k$ and $b_1 b_2 b_3 \ldots$ that appear one after another in the dictionary, is it likely or unlikely that, say, $a_j a_k b_1 b_2$ is a word anywhere in the input? The artificial mathematical binary sequences of Sections 5.2 and 5.4.1, and the DNA file results plotted in this section,

* The spike has a value of 70.8%. When the 4% dictionary is subtracted, we get an output size of 140 MB, or about 17.5 million encoded phrases. The absolute minimum that 17.5 million phrases could be encoded into in `vbyte` format has two bytes per phrase, or 35 MB; with the dictionary added on, this is a compression ratio of 20.7%, while we see 7% at the same point in Figure 5.8.

are an example of a case where this is likely. This is partly due to the alphabet being so small (with size 2 or 4), and thus it is likelier that all $n$-grams (for reasonably small $n$, under 10 or so) are valid words in the language and appear in the input. Examples where this is not the case are natural language and source code: in English, save for "had" and "that", it is very unlikely that any word will be followed by itself, and it is unlikely that the concatenation of two randomly-split word fragments is a valid word; similarly, the formal grammars of programming languages, along with the norms of programming form and style, constrain the appearance of source code. Thus, for inputs with more structure, one would want to avoid having a sample length that is too short—dictionary aspect ratio is a useful measure for categorizing different dictionary types, but sample length is the relevant parameter.

# 6. Conclusion

We have described the RLZ compression and decompression algorithms, as well an auxiliary randomized dictionary sampling algorithm, and discussed our implementation of both. We have also tested our implementation's compression performance against the LZ77-based compressors `gzip` and `xz`, and we have investigated some dictionary construction strategies.

The RLZ algorithm is a simple, and often effective, compression algorithm, that, when carefully used, is competitive with `gzip`. Compared to `gzip`, it suffers from typically slower compression speed, high memory usage, reliance on third-party suffix array construction tools, high intermediate disk usage, and high user involvement—the two LZ77-based compressors are automatic, since they compress the text "relative to itself", while RLZ requires dictionary construction. For the casual desktop user, RLZ is not a replacement for `gzip` or `xz`, or 7-Zip, or ordinary PKZIP-based zip files, or any other common compression algorithm. However, it has its niche that it was designed for—compressing *and decompressing* individual items in a very large set of very similar documents—and in its niche, it works well.

Our suite of RLZ-implementing programs, which we call `rlztools`, was built with straightforwardness in mind, and simplicity as its goal. We hope to have achieved these goals; the `rlztools` may not be the fastest RLZ-implementing code ever written, but we hope it will serve well in its job, as well as possibly being a starting-off point for future variants. We will be publishing the source code of `rlztools` online, under the `rlztools` name and with an open-source license, on some code distribution website shortly after we submit this thesis for evaluation.

In the course of our work, we have come across further possible directions of study. Firstly, on performance: with some modifications to the `rlzparse` code, compression speed might be quickly increased, especially for byte-alphabet inputs. One lookup table might list those symbols *not* in the dictionary, thus avoiding an $O(\log |D|)$-time search for every not-in-dictionary symbol—such tables might be feasible for larger alphabets also. Another table might list the first and last suffixes that start with a given symbol, or pair of symbols, thus skipping past the slowest binary searches. Such an optimization was proposed for LZ77 parsing by Kempa and Puglisi [8].

Dictionary construction seems like an area with plenty to study. We were only able to search a small space for good random dictionary construction parameters; investigating the entirety of the space (dictionary size as one parameter and aspect ratio as another) is a clear place to continue. Non-randomly built dictionaries would also be interesting: could one come up with a good heuristic for building a well-performing dictionary of a given size? For randomly sampled dictionaries, automated parameter picking would likewise be interesting: perhaps a machine-learning model could be trained to produce reasonable dictionary parameters after analyzing the input, or a sample of the input, for a moment?

For transparency, there are a couple of shortcomings in our present work that we wish to address. Although we wrote our suite of tools to work with wide input symbols (16, 32 and 64-bit words), and we have tested them to make sure they work as intended, we have not benchmarked `rlzparse`'s performance when given such inputs. This was due to a combination of time pressures and a lack of material—we did have some 32-bit data, but no description on what it was, nor did we have a larger selection of different types of 32-bit data to perform comparisons with. We have also written support in `rlzunparse` for starting and stopping decompression at any arbitrary output indices (the "I want these 50 bytes that are at position 16,581,310" feature); although quick (not instant, but quick) decompression from any given point is one of the selling features of RLZ, we have not extensively benchmarked this either, so while we have very good reason to *believe* that `rlzunparse` is faster than its competition at decompressing from a random location, we do not *know* this.

As a final remark, we hope that the `rlztools` are useful and helpful to somebody, and we are thankful of the opportunity of being able to write them.

# Bibliography


[1] J. Alakuijala, A. Farruggia, P. Ferragina, E. Kliuchnikov, R. Obryk, Z. Szabadka, and L. Vandevenne. Brotli: A general-purpose data compressor. *ACM Transactions on Information Systems (TOIS)*, 37(1):1–30, 2018.

[2] L. P. Deutsch. DEFLATE compressed data format specification version 1.3. RFC 1951, May 1996.

[3] L. P. Deutsch. GZIP file format specification version 4.3. RFC 1952, May 1996.

[4] P. Ferragina and G. Navarro. Pizza&Chili corpus – compressed indexes and their testbeds. http://pizzachili.dcc.uchile.cl/, 2005–2010.

[5] T. Gagie, S. J. Puglisi, and D. Valenzuela. Analyzing Relative Lempel-Ziv reference construction. In *Proc. SPIRE*, LNCS 9954, pages 160–165, 2016.

[6] C. Hoobin, S. J. Puglisi, and J. Zobel. Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections. *Proc. VLDB Endowment*, 5(3):265–273, 2011.

[7] P. W. Katz. String searcher, and compressor using same. United States Patent 5,051,745, September 1991.

[8] D. Kempa and S. J. Puglisi. Lempel-Ziv factorization: Simple, fast, practical. In *Proc. 15th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 103–112. SIAM, 2013.

[9] A. N. Kolmogorov. Logical basis for information theory and probability theory. *IEEE Transactions on Information Theory*, 14(5):662–664, 1968.

[10] S. Kuruppu, S. J. Puglisi, and J. Zobel. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In *Proc. 17th SPIRE*, pages 201–206, 2010.

[11] M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer, third edition, 2008.

[12] S. J. Puglisi, W. F. Smyth, and A. H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys (CSUR)*, 39(2):4–es, 2007.

[13] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *Proc. 25th annual international ACM SIGIR conference on research and development in information retrieval*, pages 222–229, 2002.

[14] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3,4):379–423, 623–656, 1948.

[15] J. A. Storer and T. G. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, 1982.

[16] World Wide Web Consortium (W3C). *Portable Network Graphics (PNG) Specification*, third edition, October 2022.

[17] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[18] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.