Local Editing in Lempel-Ziv Compressed Data

A thesis

submitted in fulfilment

of the requirements for the degree

of

Doctor of Philosophy in Computer Science

at

The University of Waikato

by

Daniel Kolver Roodt

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

2023

Abstract

This thesis explores the problem of editing data while compressed by a variant of Lempel-Ziv compression. We show that the random-access properties of the LZ-End compression allow random edits, and present the first algorithm to achieve this. The thesis goes on to adapt the LZ-End parsing so that the random access properties become local access, which has tighter memory bounds. Furthermore, the new parsing allows a much improved algorithm to edit the compressed data.

Acknowledgements

I would like to thank my supervisors, Dr. Ulrich Speidel and Dr Vimal Kumar, for their invaluable support and feedback through this project. I would also like to thank Prof. Ryan Ko for his supervision and contributions to the early stages of my PhD journey.

Thanks to the University of Waikato School of Graduate Research for the Waikato Doctoral Scholarship, which supported my studies.

I had incredible insights and ideas stemming from conversations with Bill Rogers, Ian Witten, Gonzalo Navarro, and Mark Titchener – thank you to each of you for sharing your insights and experience with me.

Thank you to my friends and lab-mates in CROW, particularly Thye Way, Chris, Joshua, TK, Harpreet and Samuel. Your friendships and support mean a lot to me, and I am proud to have worked with and gotten to know each of you.

Thank you to my family for your love and support through all my endeavours. You have been my biggest fans, and you have no idea how much your support means to me. Thank you, Mom and Dad, for laying a strong foundation for my academic studies through home-schooling.

Finally, thanks be to my Lord Jesus Christ for bringing each person mentioned into my life.

Contents

Chapter 1

Introduction

Wherever data is stored, the problem is considered: how can we store the data efficiently? The field of information theory enables us to characterise the source of the data, and then quantify how efficiently that data can be stored. Importantly, this allows us to develop techniques that approach the limits of efficient storage: These techniques are called *data compression algorithms*.

The literature review in this thesis formally describes what data compression is, some of the techniques used to compress data, and the functions which can be performed on compressed data. Some compression algorithms facilitate pattern matching on the compressed data, and others allow random access to the raw data from its compressed form. However, no currently-used universal data compression algorithm allows the data to be edited in its compressed form: editing the compressed data involves decoding the data, performing the edit on its uncompressed form, and rebuilding the compression.

In this thesis, we distinguish between *random access* and *local access*. A compression format is randomly accessible if it is possible to decompress arbitrary symbols within a constrained number of steps. A locally accessible compression format is randomly accessible, with additional constraints on the memory requirements of performing the access.

This thesis uses these concepts to explore how to edit compressed data without decompressing and then recompressing. To work towards this aim, we focus on a class of data compression algorithms called the Lempel-Ziv (LZ) algorithms. These algorithms are the most widely-used general-purpose data compression algorithms and have been studied extensively for almost 50 years [1].

This thesis focuses predominantly on a version of LZ compression developed by Kreft and Navarro, called *LZ-End*, which is designed for random access to the compressed data [2]. This compression algorithm is the starting point of the research in this thesis, which seeks answers to the following research questions:

1. Is it possible to use the random-access properties of LZ-End to allow random edits to the compressed data?
2. Is it possible to create an effective LZ compression algorithm that also supports local access?
3. Can one create a locally editable LZ compression?

This thesis answers these research questions, by providing the following original contributions:

- The thesis proposes the first algorithm to perform random edits on data compressed using a Lempel-Ziv compression format, LZ-End, using its random access properties.
- A modification of the LZ-End compression format to allow local access. The resulting compression format is referred to as *LZ-Local* in this thesis.
- An adaptation and extension of the editing algorithm for LZ-End, so that it applies to the new LZ-Local format.
- Working implementations of the algorithms above.
- A proof that the editing algorithm for LZ-Local has a significantly reduced time complexity compared to the LZ-End editing algorithm above.
- A further proof that editing LZ-Local compression has much-reduced space requirements compared to that of editing LZ-End compression.
- A suite of 1300 tests demonstrating that the editing algorithms are correct, i.e., that following an edit, a compressed string always decompresses to the expected result.

The outline of the thesis is as follows:

- Chapter 2 introduces the fundamentals of information theory, in abstract and general terms.
- Chapter 3 introduces the Lempel-Ziv compression algorithms, discussing the main members of this family of compression formats, and the operations which it is possible to perform on the compressed data.
- Chapter 4 builds on the previous chapter, by focusing entirely on the LZ-End compression format. This chapter describes the compression in depth, including the data structures used to construct the LZ-End parsing, and how it enables random access.
- Chapter 5 takes a lateral step away from LZ compression, to discuss the methodology used to evaluate the compression and editing algorithms in this thesis. As explained in the chapter, this is a necessary diversion, as the current state-of-the-art evaluation methodologies do not lend themselves well to evaluating an editing algorithm.
- Chapter 6 presents the first novel contribution of this thesis: an algorithm to randomly edit LZ-End-compressed data. This answers the first research question.

- Chapter 7 uses insights from the previous chapter to adapt the LZ-End parsing so as to allow local access. We call this new compression format LZ-Local. Chapter 7 adapts the editing algorithm from the previous chapter to apply to LZ-Local compression and evaluates the editing algorithm. This chapter answers both the second and third research questions.
- Finally, Chapter 8 reflects on the contributions of this thesis and identifies the next steps and future work.

We conclude this chapter by defining the notation used throughout this thesis.

1.1 Notation

The notation used in this thesis falls into the following categories:

- functions,
- single variables,
- indexed variables,
- structured variables,
- mathematical notation, and finally
- complexity analysis and notation.

These are described below.

1.1.1 Functions

Functions are denoted in lowercase, followed by any arguments in parentheses. For example, to calculate the Burrows-Wheeler transform of a string, we write: `bwt`(some string).

1.1.2 Single variables

A *single variable* is denoted by an italicised, lower-case letter, such as i, x, and so forth. The meaning of the variables will be declared in context and could represent an integer or a symbol from an alphabet, for example.

1.1.3 Indexed variables

We refer to *indexed variables* using a single italicised uppercase letter. Individual variables within the indexed structure will be referred to by the same letter in lowercase, with the index value as a subscript. We apply this to notation to both integer *arrays* and *strings*, with the declaration of the array/string clarifying its contents.

Every string consists of *symbols* drawn from an ordered set called an *alphabet*. We use blackboard bold uppercase letters to refer to alphabets. E.g. $\mathbb{A} = \{\text{a, b, c, d}\}$. The cardinality of the alphabet is represented as $|\mathbb{A}| = 4$.

Arrays and strings are always indexed from 0.

For example, to refer to a string of three symbols, we write $S = s_0 s_1 s_2$.

To denote a range of index values, we use comma-separated index values. For example, $t_{0,n}$ refers to the sequence of $n + 1$ variables, starting at t_0.

When we explicitly write out a full array of integers, we write the array as a comma-separated list enclosed in curly braces: e.g. $A = \{5, 8, 9, 2, 1\}$.

When we explicitly write out a string, we simply write out the string as it appears. For example, we would write $S =$ abracadabra rather than writing the individual characters as a comma-separated list in curly braces. In this case, the strings will be explicitly declared as such in their context.

Each array will have a property representing the number of elements in the array: $\Phi.size$ represents the number of elements in array Φ, and $S.size$ represents the number of symbols in string S.

We use the double-pipe characters to represent string or array *concatenation*. For example, if we have two strings S and T of length n and m respectively,

$$S \mid\mid T = s_0 s_1 \dots s_{n-1} t_0 t_1 \dots t_{m-1}$$

Finally, we use $\emptyset$ or $\{\emptyset\}$ to refer to a string or array of zero length, respectively.

1.1.4 Structured variables

Sometimes we want to refer to *structured variables*. This means that we have a single variable, which has a set number of components, each of which has its own meaning. We refer to a structured variable using lowercase Greek letters, and enclose its comma-separated components in angular brackets $\langle , , \rangle$.

For example, we represent a Lempel-Ziv phrase as π and say that this phrase is of the form $\langle b, l, s \rangle$. We use dot notation to refer to individual components of this phrase: e.g. $\pi.b$, $\pi.l$ and $\pi.s$ refer to the respective components of π.

We can have an array of structured variables, for example, $\Pi = \pi_{0,n-1}$.

Finally, we use the Greek letter Λ to refer to a *binary search tree*.

1.1.5 Mathematical notation

We use $\lfloor x \rfloor$ to refer to the floor of x, i.e., the largest integer less than or equal to x. Similarly, $\lceil x \rceil$ refers to the ceiling of x, i.e., the smallest integer greater than or equal to x.

We use the superscript -1 to refer to the inverse of a function. For example, if $f(x)$ is a function, then $f^{-1}(y)$ is its inverse: $f^{-1}(f(x)) = x$.

We use the Greek letter "rho" to represent the compression ratio of a string: $\rho(S) = 0.5$ means that string S was compressed to half its size.

All logarithms in this thesis are base-2. Because of this, we leave the base implied: that is $\log x = \log_2 x$.

We use the convention that

$$0 \log 0 = 0$$

.

1.1.6 Complexity analysis

We use the conventional definitions for the three sets of functions: Θ, O, Ω:

$$O(g(n)) = \{f(n) : \exists\, c, n_0 > 0, \text{ such that } 0 \leq f(n) \leq c \times g(n) \,\forall\, n \geq n_0\}$$
$$\Omega(g(n)) = \{f(n) : \exists\, c, n_0 > 0, \text{ such that } 0 \leq c \times g(n) \leq f(n) \,\forall\, n \geq n_0\}$$
$$\Theta(g(n)) = \{f(n) : \exists\, c_1, c_2, n_0 > 0, \text{ such that } 0 \leq c_1 \times g(n) \leq f(n) \leq c_2 \times g(n) \,\forall\, n \geq n_0\}$$

Chapter 2

Background

This chapter introduces the concepts of information theory and data compression which are relevant to this thesis. The goal of this and the following chapter is to provide the reader with a sufficient background in data compression in order to understand the remainder of this thesis.

2.1 Entropy

Claude Shannon is generally recognized as the founder of the field of information theory; however, he was in part inspired by the work of Ralph Hartley and Alan Turing. Hartley proposed the first quantitative measure of information [3]. During World War II, Alan Turing and I. J. Good built on Hartley's work while developing their cryptanalytic attack on the Enigma encryption machines. They developed a unit of measure called the *hartley*, where one hartley represents the information gained by correctly guessing one decimal digit of an enigma encryption key. The design of the Enigma machines meant that each decimal digit of the key which was correctly guessed brought more order to the encrypted message, and the cipher text therefore could be decrypted to an intermediate text which was less random than the cipher text.

During World War II, Turing visited the US Naval cryptanalysts where Shannon worked. This likely inspired Shannon to develop his theory, which was completed by the end of the war. When his work was declassified, Shannon published *The Mathematical Theory of Communication* [4], which is the seminal work in the field of information theory.

Before presenting Shannon's key results, we first need to work through a few definitions presented by Shannon in [4]:

Definition 2.1. An information *source* produces a message or sequence of messages, where the messages share common properties.

Definition 2.2. A *message* may be of various types. For example:
(a) A string of symbols from an alphabet, such as text messaging over SMS, or
(b) one or more functions of time, such as analogue radio or television signals, or
(c) several functions of several variables, such as a 3-dimensional virtual reality environment, which changes over time.

Definition 2.3. A *discrete source* is a source where both the message and the signal are a sequence of discrete symbols. E.g. digital computers use the binary alphabet $\mathbb{A} = \{0, 1\}$.

Definition 2.4. A *continuous source* is one in which the message and signal are both treated as continuous functions, e.g. radio or analogue television signals.

Definition 2.5. A *mixed source* is one in which both discrete and continuous variables appear.

Since computers process discrete information (bits), this thesis focuses on discrete sources of information: specifically, strings of symbols from a discrete, fixed-length alphabet (the first type of message listed in Definition 2.2). Throughout this thesis, we use the term *string* or *text* to refer to a message of this type.

The key results for discrete information sources outlined in Shannon's book can be summarized as follows:

- The entropy of a discrete source is defined to be

$$H = -\sum_{i=1}^{n} p_i \log p_i$$

  where there are n many possible symbols, and p_i represents the probability of the i^{th} symbol occurring. H may be interpreted as the average number of bits required at a minimum to uniquely encode each symbol in the message.

- **The Fundamental Theorem for a Noiseless Channel:**
  Let a source have entropy H (bits per symbol) and a channel have a capacity C (bits per second). Then it is possible to encode the output of the source in such a way as to transmit at the average rate $\frac{C}{H} - \epsilon$ symbols per second over the channel where ϵ is arbitrarily small. There is no method of encoding which gives a higher transmission rate than $\frac{C}{H}$, without increasing the error rate ϵ accordingly.
- **The Fundamental Theorem for a Discrete Channel with Noise:**
  For a discrete channel with capacity C and a discrete source with entropy H, if $H \leq C$ there is a coding system such that the output of the source can be transmitted over the channel with an arbitrarily small frequency of errors. If $H > C$, it is possible to encode the source so that the error is less than $H - C + \epsilon$, where ϵ is arbitrarily small. There is no method of encoding which gives errors less than $H - C$.

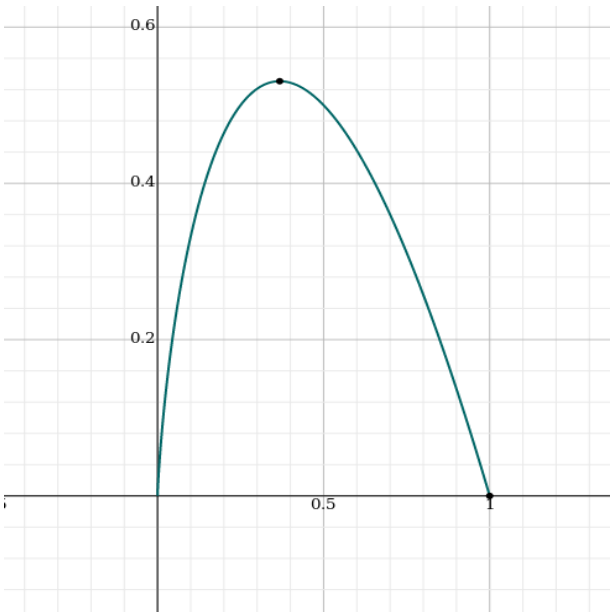


Figure 2.1: The function $f(x) = -x \log x$

The definition of the entropy of a discrete source shows that the minimum number of bits to encode each symbol depends on the probability of that symbol occurring. As can be seen in Figure 2.1, the value of $-x \log x$ attains a minimum when x is either close to 0 or close to 1. This means that the entropy will be minimized when the probabilities of each symbol occurring are either small (close to zero) or large (close to 1). When we are able to accurately predict the next symbol, the probability of that symbol will be high (close to 1), and the probabilities of all other symbols will be low (close to zero). Predictable patters in the data thus minimise its entropy. This lower entropy means that we are able to use fewer bits to encode each character, and we can

therefore store the data more efficiently. Thus, our ability to predict patterns in the data determines our ability to compress the data.

2.1.1 Empirical entropy

The above definitions provide interesting insights from a theoretical perspective. However, they have practical limitations: They assume infinite context and length of the strings. This is clearly not the case in any real-world examples.

Therefore, we also have definitions for the *order* of the entropy. The k^{th}-order entropy H_k measures the entropy of a source when we have $k \geq 0$ symbols of context.

In the base case, H_0 measures the entropy of a source with no context. This is equivalent to counting the frequency with which each symbol in the alphabet appears in the string, and using these frequencies as the only source of information to guess the probability of the next symbol in S.

Definition 2.6. Let S be a string of n symbols taken from the alphabet $\mathbb{A} = \{\alpha_0, \alpha_1, \ldots \alpha_{h-1}\}$, and let n_i denote the number of occurrences of symbol α_i in S. Then the zeroth order entropy is defined [5] as

$$H_0(S) := -\sum_{i=0}^{h-1} \frac{n_i}{n} \log\left(\frac{n_i}{n}\right)$$

Now, in practice, we usually use (a finite) context to predict the next symbol in S:

Definition 2.7. Let S be a string of n symbols taken from the alphabet $\mathbb{A} = \{\alpha_0, \alpha_1, \ldots \alpha_{h-1}\}$. Then for any string T (whose symbols are also taken from $\mathbb{A}$), let n_{Ta} denote the number of occurrences in S of the string T followed by symbol $a \in \mathbb{A}$. Let $n_T := \sum_i n_{Ta_i}$. Then the k-th order entropy of the string S is defined [5] as:

$$H_k(S) := \frac{-1}{n} \sum_{T \in \mathbb{A}^k} \left(\sum_{i=0}^{h-1} n_{Ta_i} \log \frac{n_{Ta_i}}{n_T} \right)$$

This definition says that the value $H_k(S)$ is the best compression ratio that can be achieved when a string S is compressed with k symbols of context. No compression algorithm can do better than this.

The optimal compression algorithm will therefore use $-\log \frac{n_{Ta}}{n_T}$ bits to compress symbol a when it appears following the length k context T. This definition leads to the following theorem [6, 5]:

Theorem 2.1. For any string S parsed into w distinct phrases:

$$w \log w < S.size \times H_k(s) + w \log \left(\frac{S.size}{w} \right) + \Theta(w)$$

If the string S is parsed into phrases which are not unique, the above result does not hold. However, if each phrase appears at most m times, we get [6, 5]:

Theorem 2.2. For any string S parsed into w phrases, where each distinct phrase is repeated at most m times:

$$w \log w < S.size \times H_k(s) + w \log \left(\frac{S.size}{w} \right) + w \log m + \Theta(w)$$

2.2 Defining data compression

Shannon's formulae provide us with a target to aim for: if we know the properties of our source, we know the optimum number of bits which can be used to encode each discrete symbol. But how do we go about representing the data in this optimal form? This is the problem which data compression attempts to solve.

Definition 2.8. *Data compression algorithms* parse a stream of discrete symbols from an alphabet $\mathbb{A}$ of fixed size, and produce an output which attempts to represent the input stream using fewer bits.

The parsing must be reversible: This means that the input stream must be recoverable (possibly with some errors) from the parsed output.

Definition 2.9. *Universal data compression algorithms* do not make any assumptions about the properties of the input stream, but can apply to any source.

Note that Definition 2.8 is more specific than it needs to be: continuous sequences (such as analog audio) can also be compressed. However, for this thesis, recall that we are interested only in digital information, where the information sources produce streams of symbols from a finite alphabet. Therefore, this suffices as a definition of data compression, as used in this thesis.

2.2.1 Lossy vs. lossless compression

In addition to the distinction between universal and format-specific compression algorithms, there is another factor distinguishing compression algorithms. This is the ability of the algorithm to fully recover the original data from the compressed format of data. *Lossless* compression algorithms allow the retrieval of the exact original data from the compressed form.

That is, for any valid input string S,

$$d(c(S)) = S$$

where $c(S)$ is the compressed representation of S, and $d()$ is the decompression function.

A lossless compression algorithm can compress and decompress data any number of times without producing any corruption in the string. Examples where this is important include textual data or binary code, where corruption of the data is not acceptable.

In some applications, however, retrieving the exact string is not important. An example may be image data, where some corruption can occur without causing any trouble. In fact, making small changes to an image may allow significant compression gains. This is exactly what the JPEG *discrete cosine transform* does. The design of this algorithm allows loss of data to achieve maximum compression of the image with minimal noticeable loss of image quality [7]. Such algorithms are aptly called *lossy compression algorithms*.

One downside of lossy compression algorithms is that repeatedly decompressing and recompressing the data will result in significant deterioration in the data. In the JPEG example, doing so will result in an image of lower quality each time the image is recompressed. However, this is not usually an issue for images, since the recompression only needs to happen when editing the data. Since the data often remains unedited, or is only edited once, this is often not an issue. For this reason, professional photographers often edit the raw image data before it has been compressed. They then compress the images only after the edits have been made.

It is important that a lossy compression algorithm has a bound on the errors which it introduces. We do not discuss this further, as this thesis focuses on universal, lossless compression of messages from discrete sources.

2.3 Computing the complexity of a string

Shannon's entropy laws have shown us that the size of the optimal representation of a string is tied to the string's entropy. Here, entropy can be thought of as

the string's *complexity*. It turns out that computing the complexity of a string can give us insight into computing the compressed representation of a string.

2.3.1 Kolmogorov complexity

Kolmogorov defined the complexity of a string as "the length of the shortest program which can generate the string" [8]. This is not a computable measure of string complexity; however, it does allow us to intuitively understand what the difference is between a "complex" string and a "simple" one.

This measure of complexity also highlights an important point: Increasing the length of a "simple" string does not affect its complexity. For example, the string `abcabcabcabc` has (almost) identical complexity as the string `abcabcabcabcabcabcabcabcabcabcabc`. The only difference in complexity is that we would expect the second string to require more bits to represent the number of repeated `abc`'s.

Again, this has not answered our underlying question of "how do we compute this?". However, it does highlight the utility of Lempel-Ziv complexity, which is computable.

2.3.2 Lempel-Ziv complexity

Lempel and Ziv developed an algorithm which *parses* a string and outputs a deterministic measure of its complexity [1]. This was a monumental achievement, as it allowed us to compute the complexity of arbitrary strings.

The LZ parsing algorithm takes as input a string S of symbols from an alphabet $\mathbb{A}$, and produces as output an array of *phrases* Π. Each phrase represents one or more symbols of S, and consists of:

- a back-reference pointer $b \in \mathbb{N}$,
- a length $l \in \mathbb{N}$, and
- an innovation symbol $i \in \mathbb{A}$.

The back-reference b points back to a previous symbol, the length l refers to the length of the string pointed to by the back-reference (possibly of length 0), and the innovation symbol builds upon the sequence of symbols referred to by the $\langle b, l \rangle$ pair.

Given a string S of n symbols, let's say that the first $i < n$ symbols have been parsed into $j \leq i$ phrases $\pi_0 \pi_1 \dots \pi_{j-1}$. Then the phrase π_j is constructed by finding the longest prefix of $s_{i,n-1}$ which starts in the previous part of the string $s_{0,i-1}$. Say that the longest prefix starts at symbol s_k, where $k < i$, and that the length of the phrase is $m < n - i$. In this case, the phrase $\pi_j = \langle i-k, m, s_{i+m} \rangle$. We then set $i := i+m+1$, increment j by 1, and proceed until $i = n$. Note that the back-reference component of the phrase, $i - k$ is a

relative offset, which counts the symbols in S between the start of the current phrase being encoded, and the start of the back-referenced substring.

The length component l of the LZ phrase may be 0, in which case no previous symbols are referenced, and the phrase represents only the symbol stored in the i component of the phrase. This occurs exactly once for each distinct symbol in the text. After a distinct symbol has already been processed by the LZ parsing algorithm, any occurrence of that symbol will have at least one symbol to back-reference.

For an example of constructing the LZ parsing, see Section 3.1.

The innovation symbols of the LZ phrases allows the parsing to "learn" previously-unseen symbols of the alphabet. This "learning" is done without any prior knowledge of the source of the string. This generality makes the LZ parsing algorithm universally applicable to strings from any source, with symbols drawn from any alphabet.

This parser matches our definitions of entropy: The $\langle b, l \rangle$ pairs point back to previously-seen "contexts", while every innovation symbol is a symbol in a new context.

Lempel and Ziv defined the *complexity* of a string as the number of phrases which a string S is parsed into. This is a number $\leq S.size$.

Because the LZ parsing of a text provides a measure of the text's complexity, it can also be used to generate a text of pseudo-random characters, with calibrated complexity [1]. chapter 3 discusses how to use this as the basis for compression.

2.4 Optimal compression

Definition 2.10. A parsing algorithm is *coarsely optimal* if the compression ratio $\rho(S)$ differs from the k-th order empirical entropy $H_k(S)$ by a quantity depending only on the length of the string and this difference approaches zero as the length increases [2, 5]. More formally:

$$\forall\ k\ \exists\ f_k, \lim_{n \to \infty} f_k(n) = 0,$$

such that for every string S,

$$\rho(S) \leq H_k(S) + f_k(S.size)$$

This is an important definition, and an important trait for a compression algorithm to have. How quickly the function $f_k(n) \to 0$ as $n \to \infty$ is important for practical applications, and will differentiate the performance of two coarsely

optimal algorithms. As a general rule of thumb, long strings will compress better than short ones: This is because the longer the string is, the more likely the function $f_k(n)$ is to approach 0.

2.5 Types of compression

This section introduces two families of compressors: statistical coders and dictionary-based compressors. Note that the two kinds can be "chained": for example, it is common to parse a string with a dictionary-based compressor, and then represent that compressed output more concisely using a statistical coder.

2.5.1 Statistical coding

Statistical coding algorithms use properties of the input message to choose succinct encodings for frequently-occurring symbols. For example, if the source of our message is English text, we know that the letter 'e' will occur much more frequently than the letter 'q'. A statistical encoder will therefore use a concise encoding of the letters 'e', and a more verbose encoding to represent the infrequent occurrences of 'q'. Examples of statistical encoders include arithmetic coding [9], Huffman coding [10], prediction by partial matching (PPM) [11] and asymmetric numeral systems [12].

Statistical coders can be *static* or *adaptive*. Static coders use a table which define the statistical properties of the message source. Adaptive coders may have some initial properties, but will update this as the message is parsed. Adaptive coders can therefore be effectively applied as universal compressors.

2.5.2 Dictionary-based compression

Another major class of compression techniques is to replace substrings of your text with references to a dictionary of strings. These are called *dictionary compressors*. The dictionary can be static (where the dictionary is pre-populated with substrings which we expect to frequently occur), or adaptive. In the adaptive case, the dictionary gets populated with frequently-occurring substrings as we process the text we wish to compress.

A large class of dictionary compressors are *grammar-based compression functions*. These functions replace the text with a context-free grammar which generates the text we wish to compress. The goal is that the grammar is represented by fewer bits than the string itself, in which case we have compressed the string. Commonly the grammar is then further encoded using, for example, arithmetic coding. Many universal compression algorithms are

grammar-based, such as the Lempel-Ziv family of compressors [1], Sequitur [13] and Re-Pair [14] compression algorithms, and more.

2.6 Using compressed data

Once the data has been compressed, what can we do with it? That is, what operations can we carry out on the compressed data, and in what situations do we need to decompress the data in order to perform the desired operations? In the cases where we need to decompress the data, how much of the data do we need to decompress? This section summarises the current literature's answers to these questions.

2.6.1 Compressed pattern matching

The first operation we consider is pattern matching. Do we need to decompress the string in order to search for a substring?

This problem was first solved by Amir and Benson in [15], whose algorithm applied to run-length encoding (a naïve algorithm not discussed in this thesis). Many methods to pattern match in compressed strings quickly emerged, applying to many different compression algorithms: Lempel-Ziv compressors [16, 17, 18, 19], Huffman coding [20], and Sequitur [21], to give just a few examples.

2.6.2 Random access codes

Many compressors require decoding the text sequentially from start to end: This is because decoding any symbol from the compressed data requires decoding potentially all symbols which precede it. Codes which allow *random access* do not have this constraint.

Definition 2.11. *Random access codes* allow decoding of an arbitrary symbol of text with limits on the number of other symbols which need to be decoded or accessed in order to retrieve the desired symbol.

2.6.3 Difference between local decoding and random access

In this thesis, we differentiate between *local decoding* and *random access*, where *locally decodable codes* are random access codes with the added constraint that the symbols which need to be decoded in order to decode our target symbol are within a limited distance (measured in numbers of symbols) of the target

symbol. This means that there is a specified maximum amount of memory needed in order to access the encoded symbol, so that the whole compressed text does not need to be loaded into memory to perform the decoding (which may be required in the case of random access codes). The codes defined in [22, 2, 23, 24] are all random access codes.

In addition, we apply the same distinction between *randomly editable codes* and *locally editable codes*: randomly editable codes are codes that can be edited in some time bound that is less than reconstructing the entire code. Locally editable codes are randomly editable codes with the added constraint that any part of the code which must be edited lies within some fixed distance of the position in the code which we are editing.

There are many randomly-accessible compressed codes, and Chapter 3 discusses specific examples which are relevant to this thesis.

In addition to specific examples, there are some notable results, summarised below:

- Jansson *et al.* developed a model for compressed random-access memory, called CRAM [25]. This structure uses two code tables to compress the string, and maintains a data structure over this compressed string to support efficient random access, as well as facilitating replacement, insertion, and deletion of symbols.
- Tatwawadi *et al.* provide a general scheme to convert any universal compressor so as to allow finite random access in constant time [23]. This work built on that of Mazumdar *et al.* [26]. While interesting from a theoretical perspective, the constant access time is fairly large. The scheme achieves random access by indexing and breaking the input string into a series of blocks, each of which is compressed (using any universal compression algorithm) separately of the other blocks.
  Breaking up the input text into blocks like this has a negative impact on the compression, and this negative impact is related to the size of each block: Recall the formula in Definition 2.10, and how the compression ratio $\rho(S)$ is related to the entropy of the source *plus some function inversely related to the length of the string we are compressing.* Therefore, to get satisfactory compression, each block must be large. However the constant term of the access operation cost is related to this block size.
  The novel aspect of this paper is therefore the data structure which breaks up the input message into blocks, and allows constant-time access to the index values of the symbols contained within each block. This is achieved using a compressed bit-vector of Buhrman *et al.* [27].
- Bille *et al.* showed that any grammar-based compressor can be modified to allow logarithmic-time random access to the compressed data [28].

2.6.4 Editing compressed data

Zhou *et al.* developed a method using *length-prefix compression* which allows efficiently adding to, reading from, or editing arbitrary sequences of a string independently of the remainder of the string [29]. However, this involves treating the string as a number of sequences, each of which is compressed independently of the others. The innovation is in how the sequences are stored in a neighbour-based scheme which allows efficiently editing one sequence (possibly increasing its size).

Vatedka and Tchamkerten provide a method which allows local decoding *and updating* of compressed data [30, 31]. This is similar to the work of Tatwawadi *et al.*'s random access method, and Zhou *et al.*'s method, in that it is based on splitting the input string into blocks of fixed length.

2.6.5 Compressed text index structures

A text index is a structure which attempts to store a string, so as to facilitate fast pattern matching. A compressed text index is one where the structure is stored in a compressed format. One of the most important compressed indexes is the FM-index, created by Ferragina and Manzini [32].

The FM-index, which stands for "Full-text index in Minute space", was the first text index where the space requirements were a function of the entropy of the text. The FM-index can store a text T of length n in $O(H_k(T)) + o(n)$ bits, and support searching for the x occurrences of a pattern P of length m with a time complexity of $O(m + x \log^{\epsilon} n)$ for any fixed $\epsilon > 0$.

Compressed text indexing is a very rich field [33, 34, 35, 36]. Some index structures allow edits to be made to the data [37]. However, for this thesis, we do not discuss it further, since the compressed index structures use methods of breaking up the text prior to compressing, similar to the work of Vatedka and Tchamkerten, and Tatwawadi *et al.* mentioned above.

In this thesis, we are looking at editing data in compressed form, where the compression function supports the edits, *without additional data structures imposed on top of the compressed representation.*

2.7 String attractors

Kempa and Prezza defined a *string attractor*, which is "a subset of the string's positions such that all distinct substrings have an occurrence crossing at least one of the attractor's elements" [38]. The authors showed that dictionary-based compressors approximate the smallest possible attractor of a given string, and show that string attractors support random access in optimal time.

2.8 Conclusion

This chapter has introduced data compression concepts, in the most general sense. We have discussed functions that can be applied to compressed data, and the limitations of these functions.

In the next chapter, things get more concrete: we introduce the Lempel-Ziv family of compression functions, which are based on the notion of LZ-complexity (Section 2.3.2).

Chapter 3

The Lempel-Ziv family of compressors

This chapter demonstrates how the original Lempel-Ziv parsing introduced in Section 2.3.2 applies to data compression. We explain the features of the compression algorithms based on this parsing, and how they relate to the thesis' goal of locally editing compressed data.

Unlike the previous chapter, which focused on theoretical and general concepts, this chapter provides concrete examples of algorithms and their applications.

3.1 Computing the LZ complexity

The previous chapter, introduced LZ complexity as an abstract concept. This chapter starts by providing an example of how to construct the parsing. This is important, since the remainder of the thesis depends so heavily on a strong understanding of LZ parsing.

Example 3.1. Let us compute the complexity of the string $S = \texttt{abracadabra}$. Recall that this means representing S as phrases of the form $\langle b, l, i\rangle$. We build each phrase by searching for the longest prefix of the as-yet uncompressed part of the string, which starts in the so-far compressed part of the string.

For our example string: When we parse the first phrase, we have no symbols to back-reference. Thus, the back-reference and length components of the first phrase are both 0, and the innovation symbol component is the first symbol of our string, a.

At this point, our situation is as such:

$$S = \texttt{abracadabra}$$
$$\Pi\text{: } \pi_0 = \langle 0, 0, \texttt{a}\rangle \leftarrow \texttt{a}$$

We highlight the symbols and their corresponding phrase(s) which have been processed since we last showed the state of the process.

To compress the next symbol, we are looking for the longest prefix of the unprocessed part of S which starts in the processed part of S. The start of the unprocessed part of S is a `b`, which is a symbol we have not encountered yet: again, there can be no back-reference. Likewise for the next symbol, `r`. After parsing these two symbols, our state is:

$$
\begin{aligned}
S &= \texttt{abracadabra} \\
\Pi\text{:}\ \pi_0 &= \langle 0, 0, \texttt{a} \rangle \leftarrow \texttt{a} \\
\pi_1 &= \langle 0, 0, \texttt{b} \rangle \leftarrow \texttt{b} \\
\pi_2 &= \langle 0, 0, \texttt{r} \rangle \leftarrow \texttt{r}
\end{aligned}
$$

We highlight the two phrases we have processed since the state of the process was last printed.

At this point, we have processed $s_{0,2}$, and the next symbol to parse is s_3. We have seen the symbol `a` once before, at s_0. Therefore, the back-reference of the next phrase is 3. The length is 1, since only one symbol in $s_{0,2}$ (the processed string) matches our prefix of $s_{3,10}$ (the unprocessed string). Encoding this phrase yields:

$$
\begin{aligned}
S &= \texttt{abracadabra} \\
\Pi\text{:}\ \pi_0 &= \langle 0, 0, \texttt{a} \rangle \leftarrow \texttt{a} \\
\pi_1 &= \langle 0, 0, \texttt{b} \rangle \leftarrow \texttt{b} \\
\pi_2 &= \langle 0, 0, \texttt{r} \rangle \leftarrow \texttt{r} \\
\pi_3 &= \langle 3, 1, \texttt{c} \rangle \leftarrow \texttt{ac}
\end{aligned}
$$

To parse the next phrase, we look at symbol s_5, which is an `a` again. We have now seen this symbol twice, at s_0 and s_3. Neither of these occurrences are followed by a `d`, so either can be used to encode a back-reference of length 1. We choose by default the closest option, so that we can encode the back-reference using as few bits as possible.[1]

$$
\begin{aligned}
S &= \texttt{abracadabra} \\
\Pi\text{:}\ \pi_0 &= \langle 0, 0, \texttt{a} \rangle \leftarrow \texttt{a} \\
\pi_1 &= \langle 0, 0, \texttt{b} \rangle \leftarrow \texttt{b} \\
\pi_2 &= \langle 0, 0, \texttt{r} \rangle \leftarrow \texttt{r} \\
\pi_3 &= \langle 3, 1, \texttt{c} \rangle \leftarrow \texttt{ac} \\
\pi_4 &= \langle 2, 1, \texttt{d} \rangle \leftarrow \texttt{ad}
\end{aligned}
$$

The remaining symbols to parse are $s_{7,10} = \texttt{abra}$. We have seen this sequence of symbols before, at $s_{0,3}$. We therefore encode this string as $\langle 7, 4, \emptyset \rangle$, where $\emptyset$

[1] When constructing the LZ parsing of a string, there are often multiple phrases which could be back-referenced. Although in this thesis we choose the closest phrase, this is a naïve approach. See [39] for an alternative, more concise, method of encoding back-references when there are multiple valid options.

is a symbol which does not appear in the alphabet, to show that our string ends without an innovation symbol component. Our final parsing is:

$$
\begin{aligned}
S &= \texttt{abracadabra} \\
\Pi:\ \pi_0 &= \langle 0, 0, \texttt{a} \rangle \leftarrow \texttt{a} \\
\pi_1 &= \langle 0, 0, \texttt{b} \rangle \leftarrow \texttt{b} \\
\pi_2 &= \langle 0, 0, \texttt{r} \rangle \leftarrow \texttt{r} \\
\pi_3 &= \langle 3, 1, \texttt{c} \rangle \leftarrow \texttt{ac} \\
\pi_4 &= \langle 2, 1, \texttt{d} \rangle \leftarrow \texttt{ad} \\
\pi_5 &= \langle 7, 4, \emptyset \rangle \leftarrow \texttt{abra}
\end{aligned}
$$

The parsing consists of 6 phrases, so we can say that the LZ-complexity of the string S is 6.

Remark 3.1. Note that the back-references used in this example are *relative* pointers. That is, phrase $\pi_3 = \langle 3, 1, c \rangle$ pointed to a symbol, 3 symbols prior to phrase π_3. We could have used *absolute* pointers, where the back-reference refers to the index in S of the symbol being referenced. Section 3.2 explains why we used relative back-references.

The remainder of this chapter describes prominent compression algorithms which are based on LZ parsing. From this point on, we refer to the LZ parsing as LZ-76, referring to the year that the original paper defining the LZ parsing was written [1].

3.2 LZ-77 compression algorithm

The LZ-77 compression algorithm is an adaptation of the original LZ-76 parsing. The characteristic that makes LZ-77 a compression algorithm is that it places a limit on the size of each phrase, and this limit is related to the entropy of the string being parsed.

Note that the size of the LZ parsing is affected by the number of phrases, and the size of each phrase. The size of each phrase is determined by the number of bits required to store the $\langle b, l, i \rangle$ tuples. The number of bits for the innovation symbol is constant (since the alphabet size for a text is constant). The size of the back-reference and length, however, are not constant. The number of bits required for the length of a phrase scales logarithmically with the number of symbols represented by that phrase; therefore, it is beneficial to not limit the size of the length component of a phrase. It is beneficial to limit the size of the back-reference: a longer back-reference will not add to the efficiency of the encoding compared to a shorter back-reference.

Thus, a compression algorithm built on the LZ-76 parsing will seek to maximise the efficiency of each bit required to store the back-reference. LZ-77 effectively limits the size of the back-reference component of each phrase, by restricting the search for a back-reference to a fixed-length *sliding window* of symbols which precede the current parsing position [40]. That is, for a window

size $w \in \mathbb{Z}$, all phrases' back-reference components must be $\leq w$. Recall that LZ-76, on the other hand, has no restriction as to how far back a back-reference may point.

The sliding window in LZ-77 essentially means that each symbol of a string is parsed with w symbols of context, where w is the window size. The sliding window places a constant limit on the number of bits required to store the back-reference component of the LZ phrase ($\lceil \log(w) \rceil$). This adaptation of the LZ-77 algorithm works well for compression in texts where recent patterns are likely to be more important than patterns learned further back in the parsing.

Note that for the sliding window to limit the size of the back-references, we need to use relative back-references. Absolute references would still grow logarithmically with the size of the input string.

Remark 3.2. Wyner and Ziv showed that the LZ-77 compression ratio approaches the source entropy as the window size approaches infinity [41].

Example 3.1 applies directly to LZ-77, as long as the window size is greater than 11. However, that example does not illustrate an important aspect of the LZ parsing – self-referencing phrases. This refers to the fact that while a phrase's back-reference must *start* in the already-compressed part of the text, there is no requirement that they *end* in the already-compressed part of the text: If the length of the phrase is longer than the back-reference, then the phrase starts to refer to itself. Consider the example text $S =$ `abababababab`. The first two phrases are compressed exactly as the first two symbols in our previous example:

$$S = \texttt{ab}\texttt{abababab}$$
$$\Pi\text{: } \pi_0 = \langle 0, 0, \texttt{a} \rangle \leftarrow \texttt{a}$$
$$\pi_1 = \langle 0, 0, \texttt{b} \rangle \leftarrow \texttt{b}$$

Now to reiterate, the LZ-77 compression looks for the longest prefix of the uncompressed part of the text, which *starts* in the compressed part of the text. In our case above, we are looking to compress the symbols $s_{2,9}$, and we are looking for the longest prefix of these symbols which starts in $s_{0,1}$, but which may "spill over" into $s_{2,9}$. In our example, this happens, and we compress the rest of the text in a single, self-referencing phrase!

$$S = \texttt{ab}\texttt{ababababab}$$
$$\Pi\text{: } \pi_0 = \langle 0, 0, \texttt{a} \rangle \leftarrow \texttt{a}$$
$$\pi_1 = \langle 0, 0, \texttt{b} \rangle \leftarrow \texttt{b}$$
$$\pi_2 = \langle 2, 8, \emptyset \rangle \leftarrow \texttt{abababab}$$

Each symbol in phrase π_2 references the symbol 2 index positions prior. For the first two symbols of π_2, the referenced symbols are encoded in phrases

π_0 and π_1. For the next 6 symbols, the referenced symbols occur in phrase π_2 itself. This is allowed, because we are still able to decode the text correctly, as below.

Example 3.2. Decode the LZ-77 phrases Π:

$$\begin{aligned}\Pi\text{: } \pi_0 &= \langle 0, 0, \mathtt{a}\rangle \\ \pi_1 &= \langle 0, 0, \mathtt{b}\rangle \\ \pi_2 &= \langle 2, 8, \emptyset\rangle\end{aligned}$$

Let **D** refer to the decoded text. Initially, **D** is an empty string:

$$\mathbf{D} = \emptyset$$

Decoding the first two phrases is simple – since there are no back-references, we write the symbol component of the phrases out directly, appending these symbols to **D**. The most recently decompressed phrases are highlighted:

$$\begin{aligned}\Pi\text{: } &\pi_0 = \langle 0, 0, \mathtt{a}\rangle \\ &\pi_1 = \langle 0, 0, \mathtt{b}\rangle \\ &\pi_2 = \langle 2, 8, \emptyset\rangle \\ \mathbf{D} &= \mathtt{ab}\end{aligned}$$

At this point, we have symbols $d_{0,1}$, and we wish to decompress symbol d_2. All of the information required to extract the remainder of the text is in phrase π_2. This phrase tells us that each symbol $d_i \leftarrow d_{i-2}$. This operation is to be repeated 8 times. Since $d_0 = \mathtt{a}$, we get $d_2 = \mathtt{a}$. Similarly, we copy the value of d_1 into d_3 and get $d_3 = \mathtt{b}$. Then, $d_4 \leftarrow d_2 = \mathtt{a}$, which is the first symbol encoded by our current phrase π_2. Continuing this $length = 8$ times retrieves the decoded text:

$$\begin{aligned}\Pi\text{: } &\pi_0 = \langle 0, 0, \mathtt{a}\rangle \\ &\pi_1 = \langle 0, 0, \mathtt{b}\rangle \\ &\pi_2 = \langle 2, 8, \emptyset\rangle \\ \mathbf{D} &= \mathtt{abababababab}\end{aligned}$$

3.3 LZ-SS

Storer and Szymanski optimised the LZ-77 compression algorithm by solving the problem that short LZ phrases require (sometimes significantly) more bits than the raw symbols themselves. Their adaptation of LZ-77, called LZ-SS, encodes each phrase either as a usual LZ-77 phrase or as a "run" of raw symbols [42]. The parsing uses a flag to distinguish between these two cases. A phrase consists of the form $\langle \textit{back-reference}, \textit{length}\rangle$ (note the absence of the innovation symbol) while a run of raw symbols consists of a *length* value, followed by the corresponding number of innovation symbols.

This flag adds an additional bit to the size of each phrase. However, in many use-cases this is offset by the more efficient encoding of short phrases. What is more, short phrases which are adjacent to one another can be "merged" into

one run of raw symbols, which is an additional efficiency. This is especially true when compressing the start of a string, where the first symbol will necessarily be encoded as a single phrase.

This begs the question: how does the LZ-SS parsing decide which format of phrase to use? The answer is to set a minimum phrase length m – any LZ-77 phrase whose length is less than m gets encoded as a run of innovation symbols. LZ-SS parses the LZ-77 phrases whose length is $\geq m$ as $\langle \textit{back-reference}, \textit{length} \rangle$ tuples.

Example 3.3. Let us again compress the string $S = \texttt{abracadabra}$, this time with the LZ-SS parsing, with minimum phrase length $m > 2$.

$S = \texttt{abracadabra}$

		flag	back-reference	length	symbols		raw representation
Π:	$\pi_0 =$	$\langle 0,$		$7,$	$\texttt{abracad}\rangle$	$\leftarrow$	`abracad`
	$\pi_1 =$	$\langle 1,$	$7,$	$4\rangle$		$\leftarrow$	`abra`

We see that the format of the LZ phrases has changed compared to that of LZ-77 – the LZ-SS phrases change format based on the value of the $\langle \texttt{flag} \rangle$. If the flag is 0, then the next field is a length value l, followed by a string of l symbols. If the flag was a 1, the next value is a back-reference, followed by a length l.

A notable contribution of this work is that the LZ-SS algorithm outperforms LZ-77 compression in many cases. Additionally, the LZ-SS format can be computed in linear-time (which had not previously existed for LZ-77) [42]. This is because a hash table can be used to quickly search for phrases which match the minimum required phrase length; when no match is found (in constant time), the phrase is encoded as a run of raw symbols.

3.4 LZ-78 compression algorithm

The year after Lempel and Ziv designed their LZ-77 algorithm, they designed a new version which used a dictionary, instead of a buffer. This algorithm was called LZ-78 [43]. The LZ-78 phrases are ordered pairs of the form $\langle \textit{index}, \textit{innovation symbol} \rangle$. The $\langle \textit{index} \rangle$ represents the index in the dictionary of the phrase matching the longest prefix of symbols which remain to be encoded. The $\langle \textit{innovation symbol} \rangle$ represents the symbol which comes after that longest phrase pointed to by the index.

The differences between LZ-77 and LZ-78 are therefore:

1. LZ-78 uses a dictionary, as opposed to the sliding window of LZ-77. The two are functionally similar, except for the fact that the dictionary of LZ-78 consists of codewords, which are therefore compressed. Thus, LZ-78

should be able to store the dictionary more efficiently than the window of LZ-77.

2. The length of a sequence for LZ-77 can grow very quickly, as shown in Section 3.1. LZ-78, however, can only increase the length of a pattern by one symbol at a time.
3. LZ-78 phrases cannot self-reference.

Table 3.1 shows the LZ-78 encoding of the string "abracadabra", which is the same string which we parse in Example 3.1 using the LZ-76 parsing.

Table 3.1: The LZ-78 encoding of a message

Dictionary			Symbols Encoded
index	pointer	symbol	
0	-1	∅	∅
1	0	a	a
2	0	b	b
3	0	r	r
4	1	c	ac
5	1	d	ad
6	1	b	ab
7	3	a	ra

As can be seen by comparing the above example in Table 3.1 to the encoding of the same phrase using LZ-77 in Section 3.1, the encoding of LZ-78 has the inefficiency introduced by only increasing the length of each phrase by one symbol at a time. There is the additional inefficiency introduced when a frequently occurring phrase which could be encoded as a single codeword is split at a boundary of two codewords. This can be seen in Table 3.1, where the substring "abra" occurs twice, but has still not been recognized as a phrase by the dictionary. This would not be the case in the LZ-77 algorithm, where the longest matching phrase in the preceding buffer is used as the codeword.

Another major difference between LZ-77 and LZ-78 is decoding: For LZ-77, decoding involves a sequence of array lookups. This is therefore significantly faster than compression. Additionally, decoding LZ-77 can be done in constant space, equivalent to the size of the sliding window. For LZ-78, however, decoding is just as involved as compression: both involve building the dictionary of the entire text from scratch, and the memory requirements are related to the length of the raw string.

One might ask then, why did LZ-78 become so popular? The answer is in part because the original LZ-77 paper did not present an efficient means of constructing the parsing [40]. The methods we use today to efficiently constructing the parsing, such as using a hash table [42, 44], suffix arrays [45, 35], longest previous factor array [46], or Burrows-Wheeler transform with

backward search [47], either did not exist at the time, or had not yet been applied to LZ parsing. The LZ-78 parsing, on the other hand, came with a dictionary that could be stored in a trie, facilitating fast encoding of the next phrase [43].

3.5 LZW compression algorithm

Six years after the LZ-78 algorithm was published, Terry Welch developed a similar parsing [48], called LZW. The main similarities between the two parsings were that both build up phrases incrementally, where each phrase is always 1 symbol longer than the phrase it references. However, while LZ-78 phrases consist of ordered pairs of the form $\langle \mathit{index}, \mathit{innovation\ symbol} \rangle$, an LZW phrase is simply an index.

LZW achieves this by starting with a dictionary of phrases which is fully populated with the alphabet of the source. One is able to infer which symbol is innovating on the context of a phrase by examining the first symbol encoded by the following phrase in the parsing.

3.6 Challenges when locally decoding and editing in LZ

One feature which these four types of LZ parsing (LZ-77, LZ-78, LZ-SS, and LZW) all share is that they are parsed left-to-right, in a greedy fashion. This makes local decoding and editing difficult: In the case of LZ-77 and LZ-SS, decoding a phrase involves reading any phrases which it back-references, of which there may be a large number. In the worst case, reading a phrase may involve decoding *all phrases which precede it*. In the case of LZ-78 and LZW, decoding a phrase involves reconstructing the dictionary to its state when the phrase was parsed. This means that decoding a phrase involves essentially decoding all phrases which precede it.

In all cases, editing a phrase is difficult, since *changing a single phrase may alter the string which any subsequent phrase decodes to*.

In the case of LZ-78 and LZW, if we store the dictionary/trie with the compressed parsing, we will be able to perform random access [49]. This severely degrades the compression, however, since the trie contains as many nodes as there are phrases in the LZ-78 parsing.

3.7 Searching in LZ phrases

The format of the LZ compressed phrases allows bespoke pattern matching algorithms to search over the compressed data. This is especially useful when building a *self-index*, which is a structured directory of compressed files, where each file can be decompressed independently of the others. Searching over this compressed index is very useful, as it allows the decompression of only those files which contain the desired pattern, rather than having to decompress the entire file collection.

3.7.1 Searching in LZW

The first compressed pattern matching algorithm for LZ-compressed data was developed for LZW [50]. This algorithm was the first *almost-optimal* pattern matching algorithm for an adaptive compression scheme. Its limitation was that the algorithm only found the *first* occurrence of the pattern. However, the algorithm did facilitate a tunable time-memory trade off [50].

Gasieniec and Rytter later developed an improved method for pattern matching in LZW-compressed data [17], which identified *all* occurrences of the pattern in the text. Tao and Mukherjee further improved this to support queries for multiple patterns [51].

3.7.2 Searching in LZ-77 and 78

Navarro and Raffinot then developed a general pattern-matching algorithm for LZ-compressed data [19]. The key feature of this technique is that it applies to all types of LZ parsing; the authors achieved this by abstracting away the implementation details of the LZ phrases, and developing an algorithm that applies to any "block-based" parsing. A block-based parsing is where the raw string is parsed into a series of blocks, each of which represents one or more symbols of the input string. Since any type of LZ phrase matches this definition of a "block", the pattern matching algorithm applies to all LZ parsings.

When applied to LZ-78, the pattern matching algorithm was twice as fast as decompressing and performing pattern matching on the raw data [19]. This is partly due to LZ-78 parsing being more conducive to fast pattern matching than LZ-77, and partly due to the high cost of decompressing LZ-78 compared to LZ-77's linear-time decompression.

Navarro and Tarhio later applied Boyer-Moore pattern matching to LZ-78 [52].

3.8 Random access in LZ

This section summarises the few variants of LZ parsing which allow random or local access.

3.8.1 LZB compression algorithm

The LZB algorithm was developed by Mohammad Banikazemi in 2009 [53]. At the time, it was the first LZ parsing which allowed local decoding of compressed data. Note that this is a different algorithm to the LZB algorithm described in Tim Bell's PhD thesis [54].

The LZB parsing is based on the LZ-77 parsing; the local access comes from the insight that each symbol in LZ-77 parsing is either stored in the innovation symbol component of a phrase, or references (directly or indirectly) the innovation symbol component of a previous phrase. Any symbol can be decoded as follows:

- If the symbol occurs in the innovation symbol component of a phrase, it is trivial to read the symbol directly.
- If the symbol is encoded by the back-reference component of a phrase, then identify the back-referenced phrase. Recursively decode the symbol from this back-referenced phrase; it will either be the innovation symbol of this phrase, or it will be encoded as a back-reference to a previous phrase.

The problem with this access method is that there is no tight bound on the number of recursive back-reference steps: The only bound on the number of steps in this method is the number of symbols which precede the symbol we wish to decode. For this reason, we cannot claim that the LZ-77 parsing is randomly accessible.

The LZB parsing solves this problem by maintaining a "sliding gate" over the compressed data. As LZB parses each symbol, the algorithm tracks the distance between each symbol which is encoded in the back-reference component of a phrase and its raw encoding (that is, the phrase whose innovation symbol encodes the symbol). If this distance is greater than some limit, then the algorithm finds a different encoding of the phrase. The maximum distance limit is referred to as the sliding gate of the parsing algorithm. Any encoded symbol can be decoded by accessing other symbols which are at most the sliding gate limit from the current symbol. Thus, the LZB random access is in fact locally decodable, since the number of compressed phrases required to access a symbol is constant-bounded.

Unfortunately, the LZB parsing does not compress concisely at all [2]. This means that it is of interest only from a theoretical perspective. Full credit

should be given to the author, however, for providing the first LZ parsing which is locally or randomly accessible, which enabled the work of others in this area.

3.8.2 LZ-End compression algorithm

In 2010, Kreft and Navarro developed LZ-End, another adaptation of LZ-77, which allows random access to the compressed symbols [2].

The LZ-End parsing is based on LZ-77. In fact, the definitions are identical, except that the LZ-End parsing applies the additional constraint that any back-reference ends at the innovation symbol component of a previous phrase [2]. It is this requirement which allows LZ-End to retrieve arbitrary symbols in linear time.

Kreft and Navarro used their first version of LZ-End in 2010 [2] to build a self-index [35]. Later variations of LZ-End allow parsing in linear time [55] and compressed space [56].

The LZ-End algorithm is foundational to the remainder of this thesis, and is covered in much greater deal in Chapters 4, 6 and 7.

3.8.3 Random access in LZ-78

In 2013, the LZ-78 algorithm was adapted by Dutta *et al.* to allow random access to the compressed symbols [49]. The compression ratio of the data as compared with the original LZ-78 algorithm was increased by at most a factor of $(1 + \epsilon)$, and the access time to an arbitrary symbol is $O(\log n + 1/\epsilon^2)$, where n represents the length of the compressed string.

Recall Section 3.6, which pointed out that storing the LZ-78 trie explicitly allows random access to the compressed symbols. Dutta *et al.* made use of this feature, and achieved random access by constructing a sparse Transitive Closure (TC) spanner on the LZ-78 trie. The sparsity of this transitive closure is related to the parameter ϵ in the compression ratio calculation.

The TC spanner allows us to easily infer the relationship between any two nodes in the trie. Essentially, this builds a smaller trie based on the LZ-78 parsing trie, and this trie is stored with the compressed data. This places a constant bound on the number of steps from a phrase to the nearest node which is included in the TC spanner, which can then be used to construct a short path to any other relevant nodes [49].

Sadakane and Grossi developed a separate approach to locally decodable LZ-78 compression, in the form of a concise representation of the LZ-78 trie. Storing this trie allows local decoding of arbitrary symbols [57]. The approach

used to concisely represent the LZ-78 trie has been generalised to other data structures [57].

González and Navarro obtained the same result, using a simpler scheme [58], and this result was extended in [59] to apply to the Burrows-Wheeler transform of the string.

3.9 Size of sliding window and its effect on compression ratio for LZ parsers

For LZ-77, and similar compression functions, the size of the sliding window must be chosen carefully: A large sliding window allows more possible phrase matches, and one would therefore expect to find a longer phrase match on average. However, compressing with a larger sliding window typically comes with a larger computational cost.

A larger window size also means that the back-reference pointer has a greater possible maximum value. This means that more bits are required to store the back-references for larger window sizes. We do not consider the number of bits for the length of the phrase, as the number of bits scales logarithmically with the number of symbols compressed by the phrase. Longer phrase lengths

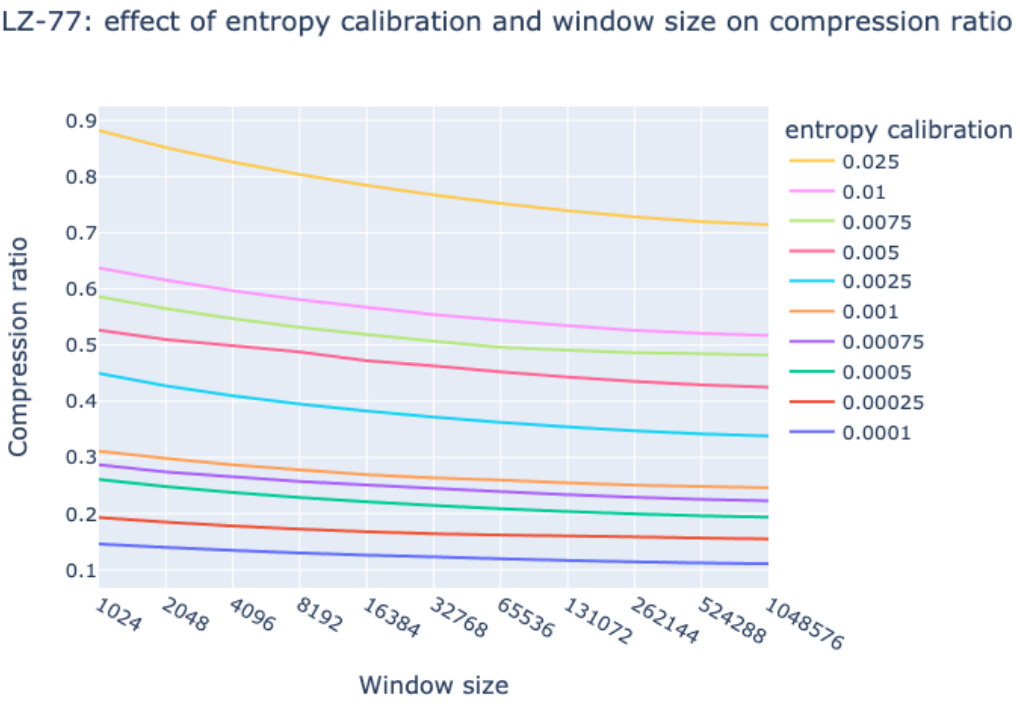


Figure 3.1: The mean compression ratio achieved by LZ-77 on strings with 10 different entropy calibrations, for various window sizes. We see that compression ratios between 10 and 90% are represented by these 10 entropy calibrations.

therefore are not expected to have a negative effect on compression ratio.

Figure 3.1 shows the effect which the size of the sliding window has on the compression ratio for files of different calibrated entropy. To demonstrate this effect, we use the set of calibrated entropy strings introduced in Section 5.2.1.

Larger window sizes make an impressive improvement to the compression of high entropy strings. This is because more combinations of symbols are occurring in the higher entropy strings, and more symbols need to be viewed before any recurring patterns appear in the same window. Lower entropy strings exhibit good compression even for small window sizes, and have rapidly diminishing returns in compression for subsequent increases in window size.

3.10 Summary of LZ parsing

Many LZ variants have not been covered in this chapter, such as LZMA, and LZ4, for example. This is because a large amount of very rich research has been invested into variations of the LZ parsing, and covering them all would take too much time. Instead, this chapter has focused on the most important variants of the LZ compressors, as they relate to the contributions of this thesis.

Chapter 4

The LZ-End parsing

This chapter describes LZ-End parsing in depth and demonstrates how the format of this parsing allows random access. The goal is to provide the reader with a basis to understand the contributions of this thesis presented in subsequent chapters. The majority of this chapter is a summary of the original paper by Kreft and Navarro, which first described this LZ compression format [2].

4.1 LZ-End parsing format

The LZ-End phrases are of the same form as for LZ-77, i.e., back-reference, length and innovation symbol tuples:

$$\langle b,\, l,\, i \rangle$$

The only difference is that the back-reference of an LZ-End phrase is a pointer to a previous *phrase*, rather than to a previous symbol.

The LZ-End parsing imposes a condition on this phrase format: The back-reference must end at the innovation symbol of a previous phrase.

Definition 4.1. (adapted from [2]) The LZ-End parsing of a string $T = t_0 t_1 \dots t_{n-1}$ is a sequence Π of $m \leq n$ phrases. If the first q phrases of Π encode the first i symbols of T, then the phrase π_q encodes the longest prefix of $t_{i,n-1}$ which is a suffix of the concatenation of phrases $\pi_{0,j}$ for some $0 \leq j < q$.

4.1.1 Example

Consider the LZ-End parsing of the text `abracadabra`:

$$T = \texttt{abracadabra}$$

$$\Pi\colon\ \pi_0 = \langle 0, 0, \texttt{a} \rangle \leftarrow \texttt{a}$$

$$\pi_1 = \langle 0, 0, \texttt{b} \rangle \leftarrow \texttt{b}$$

$$\pi_2 = \langle 0, 0, \texttt{r} \rangle \leftarrow \texttt{r}$$

$$\pi_3 = \langle 0, 1, \texttt{c} \rangle \leftarrow \texttt{ac}$$
$$\pi_4 = \langle 0, 1, \texttt{d} \rangle \leftarrow \texttt{ad}$$
$$\pi_5 = \langle 2, 3, \texttt{a} \rangle \leftarrow \texttt{abra}$$

In this example, the back-references are absolute phrase pointers; since the LZ-End parsing does not use a sliding window, absolute phrase references require the same number of bits to encode as relative references. This is in contrast to LZ-77, where the sliding window means that only relative back-references can be concisely encoded (see Section 3.2). Also, note that the back-reference points to the *last* phrase referenced – that is, we reference the sequence of l symbols which end at the innovation symbol of the phrase with index b. E.g., π_5 references the 3 symbols `abr` ending at phrase π_2.

The requirement that the back-reference ends at the innovation symbol of a previous phrase is important, as it allows fast random access to arbitrary symbols of a string: With each back-reference, we can directly read a raw symbol of the compressed text. This property allows decoding an arbitrary phrase of n symbols in at most n steps (one symbol provided by each back-reference). Section 4.5 provides the details of how this is done.

4.2 Algorithms used by LZ-End

The LZ-End parsing and its random access function use the following algorithms:

- *Backward search* [60], which itself uses:
  - a *suffix array*, and
  - the Burrows-Wheeler transform.
- a *range maximum query* on the suffix array.
- a *successor* function, implemented using a binary search tree.
- an *indexable dictionary* [61].

The remainder of this section details each of these algorithms. Section 4.3 describes how the construction of the LZ-End parsing uses these algorithms. Finally, Section 4.5 describes the algorithm for random access to symbols of an LZ-End-compressed string.

4.2.1 Suffix array

Definition 4.2. The suffix array A of a string S stores the lexicographic ordering of all suffixes of S. Each element a_i stores a pointer in S to the start of the i^{th} ordered suffix.

Example 4.1. Let S = banana\$ be our string, where the \$ denotes the end of the string and is lexicographically the lowest-ordered symbol. Then below is

the list of all suffixes of the string, along with the index in S where each suffix starts:

$$0 \leftarrow \text{banana\$}$$
$$1 \leftarrow \text{anana\$}$$
$$2 \leftarrow \text{nana\$}$$
$$3 \leftarrow \text{ana\$}$$
$$4 \leftarrow \text{na\$}$$
$$5 \leftarrow \text{a\$}$$
$$6 \leftarrow \text{\$}$$

Sorting these suffixes lexicographically yields:

$$6 \leftarrow \text{\$}$$
$$5 \leftarrow \text{a\$}$$
$$3 \leftarrow \text{ana\$}$$
$$1 \leftarrow \text{anana\$}$$
$$0 \leftarrow \text{banana\$}$$
$$4 \leftarrow \text{na\$}$$
$$2 \leftarrow \text{nana\$}$$

The suffix array A consists of the above indices in order:

$$A = \{6, 5, 3, 1, 0, 4, 2\}$$

The suffix array facilitates fast *pattern matching* because it groups all recurring substrings together. Consider in the example above both occurrences of the substring "ana", which are referenced by the third and fourth entries in the suffix array. Similarly, both occurrences of the string "na" are referenced by the last two entries in the suffix array.

The task of finding all occurrences of a pattern therefore consists of identifying the first and last suffixes which start with the search pattern. Each suffix between these two pointers represents a distinct occurrence of the pattern.

Example 4.2. Find all occurrences of the pattern "ana" in the string S = hanabanana. The suffix array of the string is $A = \{10, 9, 3, 7, 1, 5, 4, 0, 8, 2, 6\}$,

where the suffixes are:

$$
\begin{aligned}
10 &\leftarrow \$ \\
9 &\leftarrow a\$ \\
3 &\leftarrow abanana\$ \\
7 &\leftarrow ana\$ \\
1 &\leftarrow anabanana\$ \\
5 &\leftarrow anana\$ \\
4 &\leftarrow banana\$ \\
0 &\leftarrow hanabanana\$ \\
8 &\leftarrow na\$ \\
2 &\leftarrow nabanana\$ \\
6 &\leftarrow nana\$
\end{aligned}
$$

Note that the first suffix starting with the pattern "ana" is $a_3 = 7$, and the last suffix starting with the pattern is $a_5 = 5$. The suffix between these, $a_4 = 1$, also starts with the pattern. This shows that our string S contains three occurrences of the pattern "ana", and these start at s_1, s_5 and s_7.

A binary search over the suffix array can identify both the first and last suffixes starting with the pattern – thus a suffix array allows very fast pattern matching, even when the string and/or the pattern are very long.

Constructing the suffix array of a string of length n takes linear space and time ($O(n)$) [62].

Now let us generalise our pattern matching algorithm. Consider the case where we search for a pattern of m symbols P in a string of $n > m$ symbols S. A naïve pattern matching using suffix arrays will have a time complexity of $O(m \log n)$, since each suffix comparison examines m symbols. However, this has been improved to $O(m + \log n)$, by using information regarding the prefixes of successive comparisons [63]. By representing the suffix array as a binary search tree (called a *suffix tree*), pattern matching is achievable in time $O(m)$ [64]. The cost of using a suffix tree is that it is less concise than a simple array. An enhanced suffix array is a suffix array with an additional table showing the child information which is built into the suffix tree; every function allowed by a suffix tree translates perfectly to an enhanced suffix array [65], meaning that an enhanced suffix array allows pattern matching in time $O(m)$.

4.2.2 Burrows-Wheeler transform

The Burrows-Wheeler transform (BWT) of a string [66] is very similar to the suffix array, where instead of sorting suffixes of the string, it sorts the cyclic permutations of the string. The two main differences between BWT and suffix arrays are:

- The BWT of a string is itself a string. This is different to the suffix array, which is an array of index pointers.
- If one appends an "end-of-string" symbol '$' to the input string, the BWT can be constructed by a reversible function, allowing the retrieval of the original string from the transformed string. This allows the BWT to be used *in place of* the string, as opposed to the suffix array, which must be used in conjunction with the original string.

Remark 4.1. In this thesis, we use the acronym "BWT" to refer to the Burrows-Wheeler transformed string, when referring abstractly in a sentence. When referring to the transformation of a specific string in a mathematical notation, we use B as the variable to store the transformed string.

We use $B = \texttt{bwt}(S)$ to refer to the Burrows-Wheeler transform function applied to a string S. The inverse function is denoted $\texttt{bwt}^{-1}(B)$, so that $S = \texttt{bwt}^{-1}(B)$.

We explain the BWT (and its inverse) by working through examples of their construction.

Example 4.3. Let our string be S = banana$. Then the cyclic permutations, and their start indices in S, are:

$$
\begin{aligned}
0 &\leftarrow \text{banana\$} \\
1 &\leftarrow \text{anana\$b} \\
2 &\leftarrow \text{nana\$ba} \\
3 &\leftarrow \text{ana\$ban} \\
4 &\leftarrow \text{na\$bana} \\
5 &\leftarrow \text{a\$banan} \\
6 &\leftarrow \text{\$banana}
\end{aligned}
$$

We can represent this information in matrix form:

$$\begin{array}{cc} 0 & \begin{bmatrix} b & a & n & a & n & a & \$ \end{bmatrix} \\ 1 & \begin{bmatrix} a & n & a & n & a & \$ & b \end{bmatrix} \\ 2 & \begin{bmatrix} n & a & n & a & \$ & b & a \end{bmatrix} \\ 3 & \begin{bmatrix} a & n & a & \$ & b & a & n \end{bmatrix} \\ 4 & \begin{bmatrix} n & a & \$ & b & a & n & a \end{bmatrix} \\ 5 & \begin{bmatrix} a & \$ & b & a & n & a & n \end{bmatrix} \\ 6 & \begin{bmatrix} \$ & b & a & n & a & n & a \end{bmatrix} \end{array}$$

Sorting these rotations lexicographically yields:

$$\begin{array}{cc} 6 & \begin{bmatrix} \$ & b & a & n & a & n & a \end{bmatrix} \\ 5 & \begin{bmatrix} a & \$ & b & a & n & a & n \end{bmatrix} \\ 3 & \begin{bmatrix} a & n & a & \$ & b & a & n \end{bmatrix} \\ 1 & \begin{bmatrix} a & n & a & n & a & \$ & b \end{bmatrix} \\ 0 & \begin{bmatrix} b & a & n & a & n & a & \$ \end{bmatrix} \\ 4 & \begin{bmatrix} n & a & \$ & b & a & n & a \end{bmatrix} \\ 2 & \begin{bmatrix} n & a & n & a & \$ & b & a \end{bmatrix} \end{array}$$

The Burrows-Wheeler transform, B, is the rightmost column of the above matrix:

$$B = \text{annb\$aa}$$

Example 4.4. To retrieve the original string from the BWT B=annb$aa, we reconstruct the matrix of sorted cyclic permutations.

We know that the string B forms the final column of the matrix:

$$\begin{array}{cc} 0 & \begin{bmatrix} ? & ? & ? & ? & ? & ? & a \end{bmatrix} \\ 1 & \begin{bmatrix} ? & ? & ? & ? & ? & ? & n \end{bmatrix} \\ 2 & \begin{bmatrix} ? & ? & ? & ? & ? & ? & n \end{bmatrix} \\ 3 & \begin{bmatrix} ? & ? & ? & ? & ? & ? & b \end{bmatrix} \\ 4 & \begin{bmatrix} ? & ? & ? & ? & ? & ? & \$ \end{bmatrix} \\ 5 & \begin{bmatrix} ? & ? & ? & ? & ? & ? & a \end{bmatrix} \\ 6 & \begin{bmatrix} ? & ? & ? & ? & ? & ? & a \end{bmatrix} \end{array}$$

The challenge is to reconstruct the above matrix, where the '?' symbols represent the unknown letters and indices of the cyclic permutations.

We know that the rows of the matrix appear in sorted order. Sorting the

BWT string provides us with the first column:

$$\begin{array}{cc} 0 & \begin{bmatrix} \$ & ? & ? & ? & ? & ? & a \end{bmatrix} \\ 1 & \begin{bmatrix} a & ? & ? & ? & ? & ? & n \end{bmatrix} \\ 2 & \begin{bmatrix} a & ? & ? & ? & ? & ? & n \end{bmatrix} \\ 3 & \begin{bmatrix} a & ? & ? & ? & ? & ? & b \end{bmatrix} \\ 4 & \begin{bmatrix} b & ? & ? & ? & ? & ? & \$ \end{bmatrix} \\ 5 & \begin{bmatrix} n & ? & ? & ? & ? & ? & a \end{bmatrix} \\ 6 & \begin{bmatrix} n & ? & ? & ? & ? & ? & a \end{bmatrix} \end{array}$$

Since we know that the '$' symbol appears only once, at the end of the string, we know that the row with index 4 represents S. Thus, we only need to retrieve this row, rather than the entire matrix. However, we retrieve as much of the matrix as is required to retrieve this row.

From the first row, we see that an 'a' is adjacent to the '$'. This gives us an additional symbol in row 4:

$$\begin{array}{cc} 0 & \begin{bmatrix} \$ & ? & ? & ? & ? & ? & a \end{bmatrix} \\ 1 & \begin{bmatrix} a & ? & ? & ? & ? & ? & n \end{bmatrix} \\ 2 & \begin{bmatrix} a & ? & ? & ? & ? & ? & n \end{bmatrix} \\ 3 & \begin{bmatrix} a & ? & ? & ? & ? & ? & b \end{bmatrix} \\ 4 & \begin{bmatrix} b & ? & ? & ? & ? & a & \$ \end{bmatrix} \\ 5 & \begin{bmatrix} n & ? & ? & ? & ? & ? & a \end{bmatrix} \\ 6 & \begin{bmatrix} n & ? & ? & ? & ? & ? & a \end{bmatrix} \end{array}$$

We also know that there is only one 'b' in the string, and row 3 indicates that this is adjacent to an 'a'. Since our 'b' is already adjacent (by cyclic permutations) to a '$' symbol, there is only one free spot for this 'a':

$$\begin{array}{cc} 0 & \begin{bmatrix} \$ & ? & ? & ? & ? & ? & a \end{bmatrix} \\ 1 & \begin{bmatrix} a & ? & ? & ? & ? & ? & n \end{bmatrix} \\ 2 & \begin{bmatrix} a & ? & ? & ? & ? & ? & n \end{bmatrix} \\ 3 & \begin{bmatrix} a & ? & ? & ? & ? & ? & b \end{bmatrix} \\ 4 & \begin{bmatrix} b & a & ? & ? & ? & a & \$ \end{bmatrix} \\ 5 & \begin{bmatrix} n & ? & ? & ? & ? & ? & a \end{bmatrix} \\ 6 & \begin{bmatrix} n & ? & ? & ? & ? & ? & a \end{bmatrix} \end{array}$$

Looking at the BWT string B=annb$aa, we see that there are three distinct 'a' symbols in our string S. Looking at our matrix, row 0 shows that one 'a' precedes the $ symbol, and the other two 'a' symbols both precede an 'n'. In row 4, we already have the 'a' which precedes the $ sign. So, the other 'a' must precede an 'n':

$$\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{bmatrix} \$ & ? & ? & ? & ? & ? & a \\ a & ? & ? & ? & ? & ? & n \\ a & ? & ? & ? & ? & ? & n \\ a & ? & ? & ? & ? & ? & b \\ b & a & n & ? & ? & a & \$ \\ n & ? & ? & ? & ? & ? & a \\ n & ? & ? & ? & ? & ? & a \end{bmatrix}$$

Rows 1 and 2 show that the two occurrences of 'n' both precede an 'a'. Filling this in gives:

$$\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{bmatrix} \$ & ? & ? & ? & ? & ? & a \\ a & ? & ? & ? & ? & ? & n \\ a & ? & ? & ? & ? & ? & n \\ a & ? & ? & ? & ? & ? & b \\ b & a & n & a & ? & a & \$ \\ n & ? & ? & ? & ? & ? & a \\ n & ? & ? & ? & ? & ? & a \end{bmatrix}$$

There is only one unknown symbol remaining, and this is the symbol which occurs in our transformed string B which has not yet appeared in row 4 – an 'n':

$$\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{bmatrix} \$ & ? & ? & ? & ? & ? & a \\ a & ? & ? & ? & ? & ? & n \\ a & ? & ? & ? & ? & ? & n \\ a & ? & ? & ? & ? & ? & b \\ b & a & n & a & n & a & \$ \\ n & ? & ? & ? & ? & ? & a \\ n & ? & ? & ? & ? & ? & a \end{bmatrix}$$

Thus, we have retrieved our original string S=banana\$ from the transformed string B=annb\$aa.

Remark 4.2. The previous example used on many properties of the input string (such as the fact that there was only one `b` in the string) to reconstruct the string from its BWT. However, even when such assumptions cannot be made, the function $\texttt{bwt}^{-1}(S)$ is always computable in $O(n)$ time [67].

Given the similarities between BWT and suffix arrays, it is perhaps unsurprising to find that the suffix array A of a string S permits easy construction

of the BWT [2]:

$$b_i = \begin{cases} s_{n-1}, & a_i = 0 \\ s_{a_i-1}, & \text{otherwise} \end{cases} \tag{4.1}$$

This relationship, along with the suffix array's linear-time construction (recall Section 4.2.1), lets us construct the BWT in linear time and space.

4.2.3 Backward search

To demonstrate the utility of the Burrows-Wheeler transform, we use the backward search algorithm of Ferragina [60].

Let us search a string S of length n for a pattern P of length $m < n$. Then the backward search algorithm is given in Algorithm 1. The algorithm uses two auxiliary functions:

$$c_x \leftarrow \text{number of occurrences of symbols in } S \text{ lexicographically smaller than } x \tag{4.2}$$

$$\texttt{tally}(S, x, i) \leftarrow \text{number of occurrences of } x \text{ in } s_{0,i} \tag{4.3}$$

Note that one can precompute and store every possible value of Equation (4.2) in linear time and space, which is why we represent the formula as an array (C), rather than a function. The `tally` formula is represented as a function – precomputing it would require precomputing and storing a 2-dimensional array in $\Theta(n^2)$ time and space.

Algorithm 1 backward search(B, n, P, m)

Require: The BWT B of a string of n symbols
A string P of m symbols
Result: two numbers *start* and *end*, denoting the first and last index values in the suffix array which point to the pattern P.

1: $start \leftarrow 0$
2: $end \leftarrow n - 1$
3: $i \leftarrow 0$
4: **while** $i < m$ **do**
5: $\quad start \leftarrow c_{p_i} + \texttt{tally}(B, p_i, start - 1) + 1$
6: $\quad end \leftarrow c_{p_i} + \texttt{tally}(B, p_i, end)$
7: $\quad i \leftarrow i + 1$
8: **end while**
9: **return** *start*, *end*

4.2.4 Range minimum query

Recall Example 4.2: searching for the pattern $P =$ `ana` in the string $S =$ `hanabanana`, using a suffix array. The solution constructed the suffix array $A = \{10, 9, 3, 7, 1, 5, 4, 0, 8, 2, 6\}$, which showed that the pattern `ana` was referenced by $a_{3,5}$. Therefore the pattern P occurred at three locations in S, starting at s_7, s_1 and s_5.

Note that although this solution identified the locations in the string S of the pattern P, the suffix array references ($a_{3,5}$) were not ordered. This is an important limitation of using suffix arrays for pattern matching: The suffix array identifies occurrences of patterns, but these occurrences are not ordered. To find the *first* occurrence of the pattern, one must identify the smallest value a_i, $3 \leq i \leq 5$. In this small case, this is very easy. However, when the pattern occurs many times in the string, or when conducting many such queries, this is expensive.

The *range minimum query* (RMQ) is a data structure that can help in this situation [68]. This data structure is built on an unsorted array, and can identify the smallest value in any sub-array. Fisher and Heun described an RMQ structure that answers these queries in constant time [68].

In our pattern matching example from Example 4.2, we are searching for the smallest value in $a_{3,5} = \{7, 1, 5\}$. In this case, the RMQ structure will identify $a_4 = 1$ as the minimal value. Therefore, a suffix array used in conjunction with an RMQ structure can efficiently identify the *first* occurrence of a pattern P in a string S.

It may seem surprising that this query is possible in constant time. While we will not explain the details of Fisher and Heun's RMQ structure [68] here, we will briefly introduce its conceptual approach in Example 4.5, which illustrates how these constant-time queries are possible.

Example 4.5. Task: For the string $S =$ `hanabanana`, whose suffix array is

$$A = \{10, 9, 3, 7, 1, 5, 4, 0, 8, 2, 6\},$$

construct an RMQ structure over A which answers queries in constant time.

A naïve solution would be to construct a 2-dimensional array, where columns represent the start-point of each possible sub-array, and rows represent the end-point. Then each entry in the matrix represents the minimum value between the start-point corresponding to the entry's column and the end-point corresponding to the entry's row.

$$M = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{array} \begin{array}{c} \begin{array}{ccccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{array} \\ \left(\begin{array}{ccccccccccc} 10 & - & - & - & - & - & - & - & - & - & - \\ 9 & 9 & - & - & - & - & - & - & - & - & - \\ 3 & 3 & 3 & - & - & - & - & - & - & - & - \\ 3 & 3 & 3 & 7 & - & - & - & - & - & - & - \\ 1 & 1 & 1 & 1 & 1 & - & - & - & - & - & - \\ 1 & 1 & 1 & 1 & 1 & 5 & - & - & - & - & - \\ 1 & 1 & 1 & 1 & 1 & 4 & 4 & - & - & - & - \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & - & - & - \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & - & - \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 2 & - \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 2 & 6 \end{array} \right) \end{array}$$

In the matrix M, entry $m_{0,6} = 1$, because the minimum value in $a_{0,6}$ is $a_6 = 1$. However, $M_{0,7} = 0$, because the minimum value in $a_{0,7}$ is $a_7 = 0$.

This matrix supports range-minimum queries in constant time.

In the above example, constructing the matrix M is both time and space intensive. However, note that there are large blocks of 1's and 0's in the matrix – using these patterns, it is possible to construct a more concise structure to support constant-time RMQ's. Fisher and Heun achieve this by breaking the array into blocks and computing Cartesian trees (in linear time) for each block (see [68] for details). This implementation requires $O(n)$ time and space to construct.

Note that one can construct the range *maximum* query just as easily. This allows identifying the *last* occurrence of a pattern identified by the `backward search` algorithm.

4.2.5 Successor and predecessor functions

The *successor function* takes an array A and a variable x, and returns the smallest element in A which is greater than or equal to x:

$$\begin{aligned} \texttt{successor}(A, x) = a_i, \quad \text{where: } & a_i \geq x, \text{ and} \\ & a_i \leq a_j \; \forall \; j \text{ where } a_j \geq x \end{aligned} \tag{4.4}$$

A binary search tree answers this query in time $O(h)$, where h is the height of the binary tree. Therefore, a balanced binary search tree can answer the successor queries in $O(\log n)$ time, where n is the size of the array A. If, however, the tree is unbalanced, the time complexity of the successor query will be $O(n)$.

Correspondingly, the *predecessor function* returns the greatest element of A which is less than or equal to x.

Note that if the array A over which we wish to compute predecessor or successor queries is static, we can precompute these for each value, and store the result for constant-time lookups. However, in the context of LZ-End parsing, the list over which we wish to compute successor queries is dynamic; therefore, a binary search tree is an effective structure for this application.

4.2.6 Indexable dictionary

An indexable dictionary stores a bit-vector D of length n (in bits), while supporting two operations in constant time:

- $\texttt{rank}(x)$, which returns the number of 1's in D which occur in $d_{0,x-1}$; and
- $\texttt{select}(i)$ which returns the position in D of the i-th 1.

Note the relationship:

$$\forall n \geq 0, \ \texttt{rank}(\texttt{select}(n)) = n$$

The number of bits used to represent D as an indexable dictionary is dependent on its sparsity and is given below [61]:

$$\text{size of indexable dictionary} = n \log \frac{n}{x} + O\left(x + n\frac{\log\log n}{\log n}\right) \tag{4.5}$$

where n is the number of bits in the bit-vector, and x is the number of 1's (or number of 0's – whichever has fewer occurrences) in the bit-vector.

This structure is static [61], which means that it cannot be edited. To make a change to the underlying bit-vector, one must retrieve the original bit-vector from the dictionary, make the edits to the bit-vector directly, and then rebuild the dictionary. This approach is computationally expensive, and does not fit into the goal of this thesis, namely, achieving random edits in compressed data. As such, we will not store the LZ-End phrases using an indexable dictionary; we simply present it here as it is important for understanding the current state-of-the-art, and future directions to build upon the work in this thesis.

Remark 4.3. In this thesis, we define:

$$\texttt{rank}(x) := 0 \quad \text{and} \quad \texttt{select}(x) := -1 \quad \forall\, x < 0$$

4.3 The LZ-End parsing algorithm

Recall the definition of the LZ-End parsing of a string (Definition 4.1). An algorithm to construct this parsing must efficiently identify the longest prefix of the as-yet unparsed string which is a suffix of one or more successive phrases which have already been parsed.

There are multiple ways in which this can be done [56, 55, 2]; this section describes the first parsing algorithm as presented by the inventors of LZ-End, Kreft and Navarro [2].

If the first i symbols $t_{0,i-1}$ of the string $T = t_0 t_1 \dots t_{n-1}$ have been parsed into x phrases $\pi_{0,x-1}$, then at a high level, parsing the phrase π_x consists of:

1. Initialising $j \leftarrow 0$.
2. Finding all occurrences of $t_{i,i+j}$ in $t_{0,i-1}$.
3. Checking whether any occurrences of $t_{i,i+j}$ in $t_{0,i-1}$ could be a valid back-reference.
4. Keeping a reference to the first valid back-reference (if one exists).
5. If at least one occurrence of $t_{i,i+j}$ was identified in Step 2, incrementing j and going to Step 2.
6. When no occurrences are found in Step 5 (or if the end of the string is reached), encoding the phrase using the most recent back-reference stored at Step 4.

The searching (Step 2 above) is done using the backward search algorithm (Section 4.2.3). The drawback to this approach is that backward search finds all occurrences of the pattern in the entire string; in the context of LZ-End, only the occurrences of the pattern in $t_{0,i-1}$ (the already-parsed part of T) are valid. A RMQ identifies the first occurrence in T of the prefix; if this occurrence precedes t_i, it can potentially be back-referenced to form an LZ-End phrase.

Algorithm 2 formalises these steps and is explained in the following paragraphs.

The first few steps of the algorithm initialise the variables and data structures used in later steps. We describe each variable at the point it is used below.

The `while` loop at Line 4 iterates once per phrase which the algorithm outputs. Each iteration of the loop updates the following variables once:

- i, which points to the first symbol in T that has not yet been parsed into a phrase.
- p, which stores the index of the next phrase to encode.

Again, the first few steps of this loop initialise the variables used within.

The next `while` loop (Line 9) iteratively increases the length of the prefix of $t_{i,n-1}$ which is being processed into a phrase. The important variables used by this loop are:

Algorithm 2 LZ-End(T)

Require: A string T of length n symbols.
The suffix array A of the reverse of T.
The inverse suffix array I of the reverse of T.
The Burrows-Wheeler transform B of the reverse of T.
Array C, calculated using Equation (4.2), over B.
Result: An array Π of LZ-End phrases.

```
1: Λ ← [⟨−1, n⟩]
2: i ← 0
3: p ← 0
4: while i < n do
5:     s ← 0
6:     e ← n − 1
7:     j ← 0
8:     l ← j
9:     while i + j < n do
10:        s ← c_{t_{i+j}} + tally(B, t_{i+j}, s − 1) + 1
11:        e ← c_{t_{i+j}} + tally(B, t_{i+j}, e)
12:        m ← range maximum query(A, s, e)
13:        if a_m < n − i then
14:            break
15:        end if
16:        j ← j + 1
17:        x, f ← successor(Λ, s)
18:        if f ≤ e then
19:            l ← j
20:            q ← x
21:        end if
22:    end while
23:    insert(Λ, ⟨p, i_{n−(i+l)}⟩)
24:    append(Π, ⟨q, l, t_{i+l}⟩)
25:    i ← i + l + 1
26:    p ← p + 1
27: end while
28: return Π
```

- j, which stores the zero-indexed length of the prefix being processed.
- l, which stores the longest prefix which ends at the innovation-symbol component of an earlier phrase. Note that $l \leq j$.
- s and e are used (and updated) by the backward search algorithm.

Lines 10 and 11 are the steps of the backward search algorithm (recall Algorithm 1).

Recall that the range maximum query (RMQ) finds the index in the sub-array identified by the backward search steps which has the maximal suffix array value. Note from the function signature that the BWT is constructed over the *reverse* of the string T; therefore, the range *maximum* query returns

the *first* occurrence of our prefix $t_{i,i+j}$.

The backward search algorithm identifies all occurrences of a pattern within the string T; however, the only relevant occurrences of the pattern are those which have been processed already (i.e., the occurrences of the pattern $t_{i,i+j}$ which appear in $t_{0,i-1}$). Line 13 performs this check, and breaks from the loop if no valid occurrences of the pattern exist. Note that the suffix array entry a_m is compared with $n - i$ (rather than i directly) – this is again because the suffix array is constructed over the reverse of T.

If the prefix $t_{i,i+j}$ has at least one occurrence prior to t_i, the algorithm will later extend this prefix and search for a longer match. To prepare for this, increment the prefix length j.

The steps leading up to Line 17 have identified that $t_{i,i+j}$ occurs at least once in $t_{0,i-1}$. It remains to be seen whether any of these occurrences could be used as a valid back-reference, that is, whether any of these occurrences end at the innovation symbol component of a previous phrase.

A data structure (Λ) maps the phrases which have already been parsed to values in the inverse suffix array I. The structure Λ can be a binary search tree, containing ordered pairs
$\langle$`phrase index`, `inverse suffix array value`$\rangle$
The binary search tree uses the second element of the ordered pair (namely, the values in the inverse suffix array I) as its ordering – this means that the `successor`() function on Line 17 finds the phrase with the smallest value in I greater than s.

Since the variable f (Line 17) gets a value from the inverse suffix array I, this can be compared to s and e, which are index pointers into the regular suffix array A (recall that an index in A maps to a value in the inverse suffix array I). The phrases identified by the successor function will be occurrences of the string $t_{i,i+j}$ if and only if the value of f lies between s and e (the suffix array index pointers identified by the backward search step). By definition of the `successor` function, we already know that $f \geq s$. Therefore, the algorithm simply checks if $f \leq e$. If this is the case, store the length of this new phrase-encoding in the variable l, and the index value of the associated phrase in the variable q (Line 20).

The inner `while` loop (Line 9) iterates until either the end of the string is reached (Line 9) or until the pattern $t_{i,i+j}$ does not occur in $t_{0,i-1}$ (Line 13).

Upon exiting the inner `while` loop, insert into Λ a reference to the new phrase which has been constructed – recall that Λ stores the mapping between index pointers of the phrases which have already been parsed, and their positions in the suffix array A. Finally, add the newly-constructed phrase to the parsing Π.

Finally, the algorithm increments the pointer i by the length of the parsed

phrase, and increments the phrase counter p.

The process repeats until the whole string T has been parsed.

Decompressing the string works in the same way as for LZ-77, with the following minor changes to account for:

- the back-references are phrase counters, rather than symbol counters, and
- the back-references are absolute rather than relative index values.

Now that we understand how the LZ-End parsing is constructed, we move on to random access in LZ-End.

4.4 Searching in LZ-End

Recall Navarro and Raffinot's pattern matching algorithm for LZ-compressed data [19], which is summarised in Section 3.7.2. This pattern matching algorithm applies to an "block-based" parsing, which LZ-End clearly is. Therefore, we consider pattern matching in LZ-End a solved problem and do not consider it further in this thesis.

4.5 Random access in LZ-End

Random access in LZ-End is easy due to the structure of the parsing. Recall the following characteristics of LZ-End phrases:

- Each phrase contains one symbol of text (its innovation symbol).
- Each phrase points directly to the innovation symbol of a previous phrase.

These characteristics mean that each phrase provides one symbol of raw text, and each back-reference directly points to another symbol of raw text. This leads to the following important point:

> **Remark 4.4.** An arbitrary phrase encoding x symbols can be decoded with a maximum of x steps. This decoding can be done without decoding the remainder of the compressed text.

Intuitively, this is because each back-reference necessarily provides one raw symbol, so a phrase of x symbols can be retrieved by resolving at most x back-references.

Decoding a symbol t_i from the LZ parsing Π entails two tasks: identifying the phrase π_j which encodes t_i, and then retrieving the symbol from phrase π_j. For LZ-77, the first task takes time $O(j)$, and the second task has an upper time bound of $O(i)$.

LZ-End, however, achieves this in time $O(l)$, where l is the length of phrase π_j. This is achieved by mapping the symbols of T to phrases in Π (and vice-

versa) in constant time ($O(1)$) [2], and retrieving the symbols of a phrase π_j in $O(l)$.

This section describes in detail how LZ-End provides this fast random access. First, we describe how to identify the phrase which encodes a particular symbol, and then how to quickly retrieve symbols from an arbitrary LZ-End phrase.

4.5.1 Fast mapping between phrase and symbol indices in LZ-End

Recall the indexable dictionary introduced in Section 4.2.6, which supports the constant-time operations `rank` and `select`. Kreft and Navarro use these operations to create the mapping between LZ-End phrases and symbols in the raw string.

They achieve this by changing the way the LZ-End phrases are stored. Instead of storing the LZ-End parsing in the form

$$\langle b,\, l,\, i \rangle,$$

Kreft and Navarro represent the parsing array Π as three separate arrays:

- Back-references B: This array contains the back-reference component of each LZ phrase.
- Innovation symbols I: This array stores the innovation symbol component of each LZ phrase.
- Phrase boundaries dictionary D: This is a bit-vector, containing one bit for each symbol in the raw text. The bit at d_i is set to 1 if the i-th symbol of the raw text occurs at the end of a phrase, and is set to 0 otherwise. Since we expect most lengths to be significantly greater than 1 in compressible input, this is expected to be a sparse vector, the sparsity of which is related to the entropy of the text.

Kreft and Navarro represented this bit-vector D as an indexable dictionary. Recall that this dictionary facilitates the `rank` and `select` operations:

- $\texttt{rank}(i)$ returns the number of 1's which occur in $d_{0,i}$. This means that phrase $\pi_{\texttt{rank}(i)}$ encodes symbol t_i.
- $\texttt{select}(j)$ returns the position of the j-th 1 in the D array. This means that the innovation symbol component of phrase π_j represents the symbol $t_{\texttt{select}(j)}$.

The `rank` operation allows fast mapping from symbol indices to phrase indices, while `select` allows the reverse. The two operations together can be used to infer the length component of each LZ phrase:

$$\pi_j.l = \begin{cases} \texttt{select}(j) - \texttt{select}(j-1) - 1 & \text{if } j > 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.6}$$

The length component of a phrase does not include the innovation symbol, hence the subtraction of 1 for $j > 0$. Also recall that the first phrase of an LZ-End parsing is always of length 0.

Recall from Section 4.2.6 (Equation (4.5)) that the compressed structure representing the *phrase boundaries* array D can be encoded using $n \log \frac{n}{m} + O(m + \frac{n \log \log n}{\log n})$ bits, where n is the length of the uncompressed string T, and m is the number of phrases in the parsing Π. Kreft and Navarro show that this is sufficient to enable the storage of LZ-End parsing to be coarsely optimal [2] (recall Definition 2.10, which defines coarse optimality).

Note that the indexable dictionary is not the only way to map between phrases and symbols. Consider the general form of LZ parsing, where the parsing Π is represented as an array of phrases of the form $\langle b, l, i \rangle$.

In this case (which can apply to LZ-End as well as LZ-77 parsing), the time complexity of a `select`(x) operation is $O(x)$. This is because the calculation sums up the lengths of all phrases from π_0 to π_x. Similarly, the time complexity of `rank`(x) is $O(\min(x, m))$, where m is the number of phrases in Π. This is because the solution is to add up the lengths of all phrases from π_0 onwards until the sum exceeds x. This operation is $O(m)$ where m is the number of phrases in Π, which in turn is $O(x)$ as the minimum phrase length is 1.

If we expect to answer `rank` and `select` queries frequently, pre-computation can improve efficiency: Recall that `rank` and `select` compute sums over the lengths of the phrases in Π. Suppose that we precompute and store `rank`$(s * i)$ and `select`$(s * i)$ for some constant step s and $0 \leq i \leq \frac{m}{s}$. Then we can use these precomputed values to cap the number of additions in each invocation of `rank` or `select` to at most s. Precomputing the values is $O(n)$, `select` queries are $O(s)$, while `rank` queries are $O(\log(\frac{m}{s}) + s)$. The precomputed values can be stored using $O(\frac{m}{s} \log(\frac{m}{s}))$ bits.

4.5.2 Linear-time retrieval of symbols from a phrase

Let us consider the simple example where we wish to access the raw symbols from a phrase. Assume that we have already used a `rank` query to identify the phrase containing these symbols. We then retrieve the last symbol of the phrase – this is simply the phrase's innovation symbol. We then retrieve the innovation symbol of the back-referenced phrase, and the innovation symbol of its back-reference, and continue incrementally working our way backwards.

In this simple case, there is one further matter to consider: a phrase can back-reference the concatenation of one or more previous phrases. This happens when the length of the phrase we wish to recover is greater than the length of the phrase it back-references.

4.5.2.1 Example

Recall from Section 4.1.1 that the LZ-End parsing Π of the string $T =$ `abracadabra` is:

$$\begin{aligned}\Pi:\ \pi_0 &= \langle 0, 0, \mathtt{a}\rangle\\ \pi_1 &= \langle 0, 0, \mathtt{b}\rangle\\ \pi_2 &= \langle 0, 0, \mathtt{r}\rangle\\ \pi_3 &= \langle 0, 1, \mathtt{c}\rangle\\ \pi_4 &= \langle 0, 1, \mathtt{d}\rangle\\ \pi_5 &= \langle 2, 3, \mathtt{a}\rangle\end{aligned}$$

Say we wish to retrieve the symbols $t_{7,10}$. The `rank` function tells us that these are the symbols of the final phrase, π_5.

We reconstruct the retrieved string, R, from back to front, starting with the innovation symbol of π_5:

$$R = \_\_\_\mathtt{a}$$

We then process the back-reference of π_5. We know that π_5 has a length of 3; however, its back-reference (π_2) has a length of 0. This means that we retrieve one symbol from π_2 (its innovation symbol), and the remaining symbols of π_5's back-reference come from the phrases which precede π_2. In our example, we retrieve one further symbol ("`b`") from π_1, and the remaining symbol ("`a`") comes from π_0 to get our retrieved text:

$$R = \mathtt{abra}$$

This processing of concatenated phrases lends itself well to a recursive algorithm, described in the next section.

4.5.3 Random access algorithm

Algorithm 3 represents the recursive random-access algorithm hinted at in the previous section.

The algorithm's input is the LZ-End parsing Π of a string T. As before, we represent Π via three arrays: B (the back-references of the LZ-End phrases), I (the innovation symbols of the LZ-End phrases) and D, the indexable dictionary of the bit-vector which identifies the phrase boundaries.

Additional arguments to the algorithm are the index s of the first symbol in T to retrieve, and the length l of the sequence of symbols to retrieve, starting at s. For reasons that will become clear in Chapter 7, s can be negative: in this case, we first assign e (the index in T of the final symbol to retrieve), and then assign s to zero, if it was negative (Lines 1 and 2).

The algorithm proper only runs if one or more symbols will actually be retrieved (Line 4).

Algorithm 3 `random access`(Π, s, l)

Require: LZ-End parsing Π of a string T.
Recall that Π is represented as three arrays: B, I and D.
The index s of the first symbol in T which we want to retrieve.
The number of symbols l which we want to retrieve.

Result: A string $R = t_{s,s+l-1}$

```
1: e ← s + l − 1
2: s ← maximum(s, 0)
3: R ← ∅
4: if s ≤ e then
5:     p ← rank(e)
6:     if d_e = 1 then
7:         R ← Random access(Π, s, l − 1)
8:         R ← R || i_p
9:     else
10:        q ← select(p − 1) + 1
11:        if s < q then
12:            R ← Random access(Π, s, q − s)
13:            l ← e − q + 1
14:            s ← q
15:        end if
16:        R ← R || Random access(Π, select(b_p) − select(p) + s + 1, l)
17:    end if
18: end if
19: return R
```

Once the number of symbols to retrieve has been confirmed to be greater than zero, the next step identifies the index p of the phrase which encodes the symbol at index e.

The symbol at index e is either the final symbol of phrase π_p, or it is encoded by the back-reference component of phrase p. Line 6 distinguishes between these cases: the dictionary D marks phrase boundaries, so d_e will have a value of 1 if and only if the last symbol we wish to decode occurs at a phrase boundary.

If the symbol t_e is represented by the innovation symbol of phrase π_p (i.e., if $d_e = 1$), we recursively retrieve the symbols $t_{s,e-1}$ (Line 7). We then append the innovation symbol i_p of phrase π_p to the end of the string in Line 8 and return the resulting string R in Line 19.

The `else` block starting at Line 9 handles the case where the final symbol to retrieve, t_e, was not the final symbol of a phrase (i.e., if $d_e = 0$). In this case, we need to distinguish whether the symbols we wish to retrieve, $t_{s,e}$, are encoded by a single LZ-End phrase, or by the concatenation of multiple phrases. To answer this question, we calculate q, the index (into the raw string T) of

the first symbol encoded by phrase π_p (Line 10).

If $s < q$, it means that the first symbol to retrieve, t_s, occurs in a different phrase to the last symbol to retrieve, t_e. In this case, recursively use the `Random access()` function to retrieve the phrase(s) which precede π_p (the phrase encoding t_e). Then adjust the variables s and l to refer to the symbols which will not yet have been recovered following the call to the `Random access()` method. These steps are detailed in Lines 12 to 14.

When we reach Line 16, we have retrieved all symbols coming from phrases which precede π_p. We also know that the final symbol to retrieve is not the innovation symbol of phrase π_p. We therefore make a recursive call to the `Random access()` function, which retrieves the relevant symbols from the back-reference component of π_p. Recall that b_p is the entry at index π_p of the back-reference array B. Therefore the expression

$$\texttt{select}(b_p) - \texttt{select}(p) + s + 1$$

yields the index in T of the first symbol pointed to by the back-reference component of phrase π_p.

Kreft and Navarro show that the `Random access()` algorithm retrieves the symbols $t_{s,e}$ of an LZ-End compressed string with a time complexity of $O(e - s + 1)$ if t_e occurs at the end of a phrase. This assumes that the phrase lengths are stored in an indexable dictionary, so that the `rank` and `select` operations are performed in constant time. If, however, we were to store the phrases in their $\langle b, l, i\rangle$ form and use a cache to speed up the `rank` and `select` operations, the time bound to retrieve $t_{s,e}$ increases to $O((e-s+1)\times(\log \frac{m}{x}+x))$, where there are m phrases in the LZ-End parsing, and the cache stores every x-th phrase.

Note that Algorithm 3 requires the LZ-End parsing to be represented in the format described by Kreft and Navarro in [2], i.e., as three arrays B, I and D. However, with minor changes the algorithm can apply if the parsing is represented in the general LZ format, i.e., as an array of $\langle b, l, i\rangle$ tuples. In this case, the `rank` and `select` queries will not run in constant time.

4.6 Summary of the LZ-End parsing and random access

This chapter has described a possible method to construct the LZ-End parsing, and how this parsing allows random access. We have also described in detail the algorithms and data structures used to do this, and will meet them again throughout the remainder of the thesis.

The next chapter takes a small but necessary detour to explain the evaluation methodology used in this thesis, which prepares us for discussing the first of this thesis' central contributions in Chapter 6: an algorithm to make random edits to an LZ-End parsing without decompressing the entire string.

Chapter 5

Evaluation methodology

The previous chapter provided an overview of the LZ-End compression algorithm, which we build upon in this thesis. Before building on this compression algorithm, however, we must first decide how to evaluate the contributions of this thesis. This chapter reflects on the typical approach of existing research to empirically evaluate compression algorithms. We discuss the efficacy of these approaches to evaluating editing algorithms and the challenges that this presents. The chapter thus continues by describing how we will evaluate the compression and editing algorithms presented in this thesis.

5.1 Empirically evaluating compression algorithms

This section discusses existing techniques for the empirical evaluation of compression algorithms. We first discuss which factors are important, and then discuss how to go about measuring them.

5.1.1 Compression performance evaluation criteria

The main aspects in evaluating the performance of data compression algorithms are:

- The compression ratio $\rho(S)$: If string S is compressed into a parsing Π, we define the compression ratio as:

$$\rho(S) = \frac{\text{number of bits to represent } \Pi}{\text{number of bits to represent } S}$$

  How do we calculate the number of bits required to store the parsing Π? In the case of LZ-based compression algorithms, there are many different ways to store the phrases, such as using the indexable dictionary to store phrase lengths (Section 4.5.1) or bit recycling the back-references

[39]. These different formats of storing the LZ phrases will affect the compressed size of the string, as well as affecting the time to compress, decode, randomly access, or edit the data. We discuss the method used in this thesis in Section 5.3.

- Compression time complexity.
  The importance of compression speed depends to a large degree on the use case. If we are compressing data to transmit over a network, then the time taken to perform compression adds to the latency of communicating over the network. In this use case, compression speed is quite important. If, however, one compresses the data immediately before writing it to a disk for archiving, compression speed may be less important, as there is no time-critical task waiting to use the data.
  Another factor affecting how much emphasis we place on compression speed is the number of times we expect to have to compress the data compared to performing other operations. If we expect to be editing the compressed data frequently, then compression speed will be more important. If, however, we do not expect to edit the data after compression, then we place less emphasis on compression speed.
- Decompression time complexity.
  Decompression speed is very important, as we generally decompress the data immediately before using it. Decompression may often be used more frequently than compression, e.g., a software package may be compressed once, and then downloaded from a web repository many times over. For these two reasons, decompression speed is often viewed as being more important than compression speed.
- Space complexity of compression and decompression: The compression techniques can often run using different kinds of data structures, with associated time-memory trade-offs (TMTO's).
  Consider again the use case where we compress the data immediately before archival. In this case, the compression may run as a background process with low priority, since no other process depends on it. We would like the background process to use as little memory as possible so that this resource is available to other processes on the system.
  Compression and decompression often have different memory requirements; this is especially true of the LZ-77 compression function [40], for example, where compression requires a lot more processing and memory compared to decompression.

5.1.2 Compression corpora

Now that we know the important factors to measure, we need a system to compare different compression algorithms in a consistent, replicable way.

To this end, various researchers have standardised sets of files, called *compression corpora*, over the years. These generally consist of a range of files of different formats and sizes. The goal of a good compression corpus is to provide an unbiased representation of the performance of the compression algorithm across a wide range of strings, to determine where the algorithm works particularly well, and where its weaknesses are. Having an accessible compression corpus allows for the results of one algorithm to be easily understood, reproduced and compared to other works.

5.1.2.1 Calgary corpus

The Calgary corpus was the first widely used benchmark for evaluating the performance of compression algorithms. It consists of a collection of 14 files, ranging in size from 21KB to 770 KB. The corpus contains formatted and unformatted ASCII English text, a VAX executable, a bitmap image, source code written in C, Lisp, and Pascal, as well as seismic (numeric) data [69].

5.1.2.2 Canterbury corpus

The size and types of files in the Calgary corpus quickly became obsolete, as storage became cheaper, file sizes grew, and programming languages and file formats changed. Thus arose the need for a more modern corpus, which was filled by the Canterbury corpus [70]. The Canterbury corpus consists of 11 files, ranging in size from 4KB to 1MB. The data includes English and Shakespearean text, HTML, C, and LISP source code, as well as an Excel spreadsheet, a SPARC executable and fax images.

Each file in the corpus was chosen from a pool of files of the same format. The files in each format were compressed using a variety of compression methods. From this group of files, a single candidate was chosen for inclusion in the corpus. The chosen file was the one that produced the most consistent results across the various compression algorithms used, and whose compression was closest to the regression line best describing the compression of each file in the given format [70].

5.1.2.3 Pizza & Chili corpus

The Pizza & Chili corpus was developed specifically to test compressed string indices [71]. The corpus consists of several sub-collections: source code, numeric

pitch values, protein sequences, DNA sequences, English text, and XML structured text. The files themselves are much larger than those of the Calgary and Canterbury corpora, ranging from 55MB to 2.2GB. The larger amount of data in the Pizza & Chili corpus allows algorithms to converge further towards their asymptotic performance in operations such as pattern matching on compressed files.

5.1.2.4 Calibrated entropy strings

The final method we consider for quantifying the performance of compression algorithms was developed by Ebeling, Steuer and Titchener [72]. This consists of using a logistic map with noise insertion and a threshold to generate a sequence of real numbers x_i, which can be converted into a string of bits with calibrated entropy.

$$x_{n+1} = r_\infty x_n(1 - x_n) + \epsilon\xi_n, \qquad \text{where } \xi_n \in [-1, 1] \tag{5.1}$$

In the above equation, $r_\infty \approx 3.5699$ is the Feigenbaum accumulation point, ϵ is the noise amplitude, and the random numbers ξ_i are noise [72].

One then applies a generating partition with partition value c to the logistic map above to create a sequence **S** of bits as follows:

$$s_n = \begin{cases} 0, & \text{if } x_n \leq c, \text{ and} \\ 1, & \text{if } x_n > c \end{cases} \tag{5.2}$$

Varying the noise amplitude ϵ in Equation (5.1) and the partition value c in Equation (5.2) changes the entropy of the strings produced [72, 73, 74]. Section 5.2 of [72] shows the effect of changing these parameters.

This method allows the generation of arbitrary-length strings of calibrated entropy (which we do in Section 5.2). When used alongside a well-known compression corpus such as the Canterbury corpus, the logistic map can be used to extrapolate the evaluation of compression algorithms to different data sources whose entropy is known.

5.1.3 Compression corpora as a tool for evaluating universal compressors

While compression corpora are very valuable tools, their use can lead to misleading results. The compression ratio is a function of two variables: the compression algorithm and the input data (which is typically a compression corpus). We would like to compare two compression algorithms such that the

comparison is invariant under a change to the compression corpus (i.e., the input data).

We use a thought experiment to demonstrate the difficulties of comparing two algorithms such that the results are independent of the input data: Consider the case where we are comparing the compression ratio achieved by two compression algorithms, f_1 and f_2 on all strings up to length l. The results will fall into one of the following categories:

- The compression ratio achieved by f_1 matches that of f_2 for all strings. In this case, the performance of the two functions is identical.
- For each string, the compression ratio achieved by f_1 exceeds or at least matches that of f_2 (or vice versa). In this case, it is clear which is the better algorithm (for strings of length up to l).
- There exist some strings for which f_1 achieves a better compression ratio than f_2, and vice versa. This is a common case in practice (see for example Section 7.2.1).

  In this case, it is possible to construct a corpus C of q strings for which f_1 achieves a better compression ratio than f_2, and another corpus C' of q strings for which f_2 achieves a better compression ratio than f_1. Therefore, the comparison between the two algorithms is not invariant under choice of corpus.

While steps have been taken to avoid bias in the Canterbury corpus [70], it cannot be ruled out entirely, especially if the comparison results are close.

An objective criterion might be to run f_1 and f_2 on all strings up to length l and add up the compressed lengths for all such strings. If f_1 compresses the strings to a *total length* that is shorter than f_2, then f_1 outperforms f_2 on that set of strings. If this holds as l goes to infinity, then f_1 is a better universal compressor than f_2. Of course, this proof is not computable.

What such a thought experiment demonstrates is that if we have small differences between f_1 and f_2 on a corpus, we cannot conclude that either algorithm is a better universal compressor. Rather, we can only note the performance difference on the specific corpus. Evaluating compression algorithms on corpora such as those listed above can only lend evidence to support the superiority of particular algorithms.

5.1.4 Shortfalls of compression corpora when evaluating editing algorithms

Compression corpora such as the Canterbury corpus were designed for measuring compression of a string. What about the case where we wish to measure the performance of an algorithm which edits compressed strings? If we wish to

insert a string into one of the files from a corpus, we need to find a candidate string to insert, which has similar properties to the existing file. For example, say we wish to insert a string into an English text file: Is it important that the text file is grammatically correct following the insertion? How do we craft a string to be inserted into Alice in Wonderland?

A more methodical approach would be to compress a string from a given source and insert into this string another string from the same source. This should generally produce results more consistent with the notion of entropy as being a measure of a source of information, as opposed to inserting into a static string.

5.2 Evaluation methodology used in this thesis

In this thesis, we assess the time and memory performance of algorithms theoretically, and empirically measure compression ratio. The reason we do not empirically measure and report time and memory usage is twofold: First, theoretical assessments allow us to make adequate comparisons between the algorithms, and second, empirical measurements of these attributes are heavily dependent on the implementation, in addition to the characteristics of the algorithms themselves and the context in which they are run. Due to time constraints, we have not been able to implement the most time- and space-efficient versions of all algorithms involved. Compression ratio, on the other hand, is more easily measured.

To evaluate a compression algorithm empirically, we measure the compressed length of the Canterbury corpus files, in addition to a collection of calibrated entropy strings. The Canterbury corpus allows comparison with other researchers who have used this corpus in the past, as well as future research which may use it. The calibrated entropy strings allow more general comparisons, as pointed out already in this chapter. To evaluate algorithms that edit compressed strings, we only use the calibrated entropy strings. Whenever an edit involves inserting a string into the compressed string, we generate the inserted string from the same source as the string we are editing.

5.2.1 Generating calibrated entropy strings

Recall that each bit s_n of the calibrated string S is generated as follows:

$$x_{n+1} = r_\infty x_n(1 - x_n) + \epsilon\xi_n, \qquad \text{where } \xi_n \in [-1, 1] \tag{5.1}$$

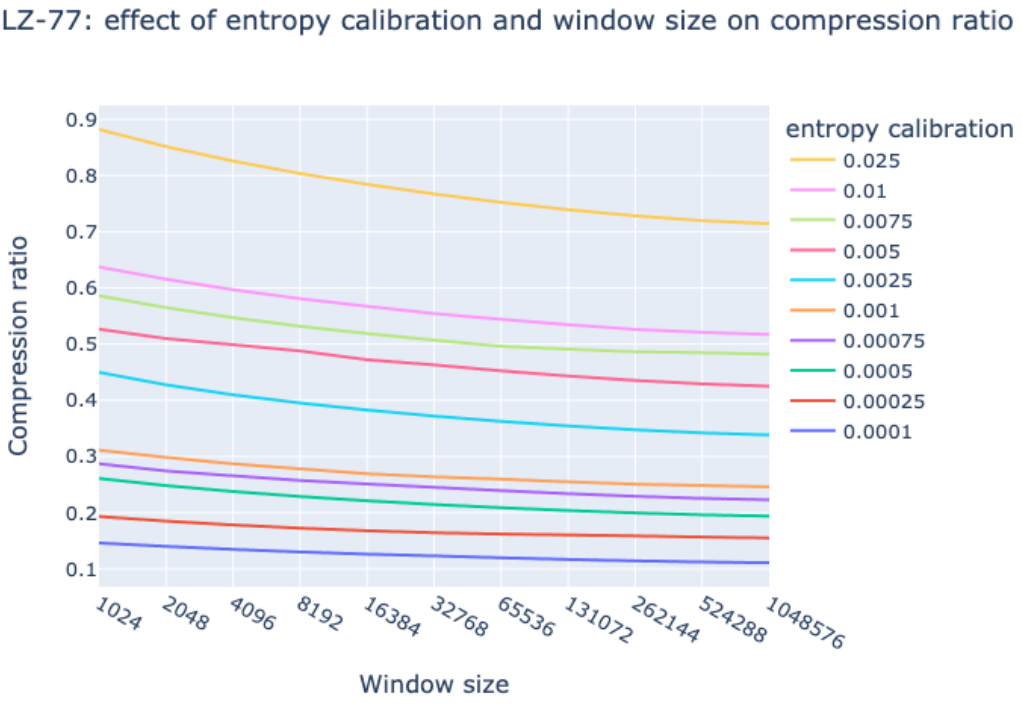


Figure 3.1: The mean compression ratio achieved by LZ-77 on strings with 10 different entropy calibrations, for various window sizes. For each entropy calibration, there are 30 strings, each of length 5MB. Note that these 10 entropy calibrations represent strings compressing to between 10 and 90% of their original size.

$$s_n = \begin{cases} 0, & \text{if } x_n \leq c, \text{ and} \\ 1, & \text{if } x_n > c \end{cases} \tag{5.2}$$

Recall also that $r_\infty \approx 3.5699$ is the Feigenbaum accumulation point, ϵ is the noise amplitude, the random numbers ξ_i are noise, and c is the partition value.

The entropy of this source is affected by the values r, ϵ and c. We keep r and c constant (≈ 3.5699 and 0.5, respecively). A cryptographically secure random number generator supplies the random values ξ_i. The seed value for each string is $x_0 = 1$. This means that x_1 depends solely on the term $\epsilon\xi_1$.

When the values r, ξ and c are set as above, then the Shannon entropy of the resulting sequence x_i is related to the value ϵ [74, 75]. We used the (somewhat arbitrarily chosen) noise amplitudes:

$$\epsilon \in \{0.0001, 0.00025, 0.0005, 0.00075, 0.001, 0.0025, 0.005, 0.0075, 0.01, 0.025\}$$

This choice of noise amplitudes provided a good range of strings achieving different compression levels (see Figure 3.1).

Note that from now on, we refer to the strings by their respective noise

amplitudes ϵ (see the legend in Figure 3.1); this is to highlight the fact that the only differentiator between the strings is the noise amplitude.

For each entropy calibration, we generated 30 'compression strings' of 5MB each. All experiments for each entropy calibration were conducted on the same set of 30 strings.

When measuring compression ratios, we report the mean compression ratio of the 30 strings for each entropy calibration. To evaluate an editing algorithm, we edit each of these compressed strings, and again report the mean results for each entropy calibration. In some cases, the edit operation inserts a string into the compressed string. To facilitate this, we have one additional "insertion string" for each entropy calibration. This means that all edits come from the same source as the string that is being edited. Note that this would not be possible if using a static corpus such as the Canterbury corpus.

5.2.2 A note on the consistency between calibrated entropy strings

The previous section begs the question: How consistent are the results between logistic map files of different entropy calibrations?

Table 5.1 answers this question, by listing the standard deviation of each data point in Figure 3.1. We see that the standard deviation is a small percentage (at most 3%) of the compressed size for any given data point. Therefore, we can conclude that these strings of calibrated entropy are producing very consistent results when evaluating the LZ-77 compression function.

5.3 Storage format for LZ phrases in this thesis

In this thesis, we wish to compare the compression ratio achieved by various LZ-based compression algorithms. This comparison is complicated by differences between the variants of LZ parsings. LZ-End, for example, supports fast random access when aided by an indexable dictionary; there is no benefit, however, to using an indexable dictionary to store LZ77 phrases. This makes a direct comparison between LZ77 and LZ-End compressed sizes difficult.

In order to make meaningful comparisons between the variants of LZ parsings, we store all LZ77-based parsings as an array, where each cell of the array is of equal size, and has the format: $\langle b, l, i \rangle$. We do not use indexable dictionaries or other data structures to aid in the parsing.

Table 5.1: Standard deviation in LZ-77 compression ratio for the experiments shown in Figure 3.1.

	Entropy calibration									
Window size	0.00010	0.00025	0.00050	0.00075	0.00100	0.00250	0.00500	0.00750	0.01000	0.02500
1024	0.0034	0.0025	0.0047	0.0088	0.0043	0.00017	0.0096	0.00012	0.00012	0.00018
2048	0.0034	0.0023	0.0043	0.0075	0.0046	0.00015	0.011	0.00010	0.000097	0.00017
4096	0.0032	0.0021	0.0040	0.0074	0.0037	0.00011	0.0095	0.000090	0.000083	0.00015
8192	0.0023	0.0019	0.0037	0.0061	0.0044	0.00010	0.0047	0.000090	0.00010	0.00014
16384	0.0020	0.0018	0.0034	0.0056	0.0037	0.000097	0.0066	0.0036	0.0040	0.00015
32768	0.0013	0.0017	0.0032	0.0050	0.0049	0.00010	0.00011	0.0034	0.000084	0.00013
65536	0.0012	0.0021	0.0030	0.0033	0.0055	0.000098	0.00011	0.00010	0.0035	0.00018
131072	0.0011	0.0018	0.0029	0.0031	0.0047	0.000094	0.00011	0.0072	0.0034	0.00014
262144	0.0011	0.0018	0.0027	0.0029	0.0044	0.000086	0.000102	0.0081	0.000091	0.00013
524288	0.0010	0.0018	0.0026	0.0028	0.0029	0.000085	0.00010	0.0069	0.0042	0.00011
1048576	0.0010	0.0019	0.0025	0.0027	0.0022	0.000071	0.000093	0.0056	0.0040	0.00011

We calculate the compressed size of the string as:

$$\lceil \log \bar{b} \rceil + \lceil \log \bar{l} \rceil + \lceil \log a \rceil \tag{5.3}$$

Here, $\bar{b}$ is the maximum back-reference component of a phrase in the parsing, $\bar{l}$ is the maximum length component of a phrase, and a is the size of the alphabet. We then multiply the number of bits required to encode each phrase by the number of phrases in the parsing.

Although not resulting in the optimal compressed size, or the fastest random access (in the case of LZ-End), this approach meets the requirements of this thesis. The primary goal of the empirical evaluation is to measure how editing the compressed data affects the compression ratio: in this case, we are comparing a compressed size prior to and after the edit. The simple compressed format which we use allows us to make these comparisons.

This format has several additional advantages:

- It allows direct and meaningful comparisons between variants of LZ compression algorithms.
- The arrays facilitate constant-time lookups.
- It is simple and easy to implement, allowing for robust, accurate experiments.

As this thesis represents the first attempt to edit LZ-compressed data, the simple format will enable future research to evaluate the costs and benefits of data structures which attempt to improveme on this storage format.

5.4 Conclusion

This chapter summarised the evaluation methodology used by most researchers who evaluate universal compression algorithms. We discussed the limitations of this methodology, especially when applied to compressed editing algorithms. We introduced the methodology which we use in this thesis, whereby we use strings of calibrated entropy, and demonstrated that these calibrated entropy strings produce consistent and replicable results. Finally, we described the consistent storage format for all LZ-based compression algorithms implemented and compared in this thesis.

Chapter 6

Random edits in LZ-End data

This chapter presents the first novel contribution of this thesis: the first algorithm to randomly edit LZ-compressed data. This editing algorithm, which has been briefly described in [76], applies to LZ-End phrases stored in the general LZ format: $\langle b, l, i \rangle$. This is as opposed to the phrases stored in a more specific format, such as Kreft and Navarro's version, which uses an indexable dictionary to store phrase lengths [2]. Editing the general phrase format has two advantages: 1. the editing algorithm applies to any format derived from the LZ-End parsing, and 2. the editing algorithm will more readily relate to other LZ compression formats.

The first section in this chapter, 6.1, defines an edit, discusses the implications of storing the compressed data explicitly as phrase tuples $\langle b, l, i \rangle$, rather than using supplementary data structures like a phrase dictionary, and finally summarises the difficulties of editing LZ-compressed data.

Section 6.2 explains the editing algorithm itself. The remaining sections of the chapter discuss implications for how the edit affects the LZ-End parsing, analyse the time and memory requirements of the editing algorithm, and discuss how editing affects random access. This chapter does not empirically evaluate the editing algorithm; the evaluation appears in Chapter 7.

6.1 Preliminary notes on editing LZ-End

This section lays the groundwork for understanding the editing algorithm presented in this chapter.

6.1.1 Format of the parsing

In their original paper, Kreft and Navarro stored the LZ-End parsing as three arrays: B, which stored the back-references, I, which stored the innovation symbols, and D, which was an indexable dictionary storing phrase boundaries

(which allows calculating the phrase lengths) [2]. Recall Section 4.5.1 for the details of this implementation. Kreft and Navarro chose this representation for the LZ-End parsing, as it allowed constant-time `rank` and `select` queries.

For this chapter, however, we revert back to the original LZ format of the parsing, where each phrase is stored explicitly as a $\langle b,\ l,\ i\rangle$ tuple. We assume that this parsing supports `rank` and `select` queries which behave identically to the same operations in Section 4.5.1, however, we do not make assumptions about the implementation or run-times of these functions. Rather, we decouple these functions from the editing algorithm, and discuss this further in Section 6.5.

6.1.2 Defining the edit operation

We define an edit operation in terms of replacing symbols at specified indices of a string. Therefore, the edit operation has four parameters: $\texttt{edit}(T, i, j, S)$, where:

- T is the string being edited,
- i is the index of the first symbol in the raw form of T which is being removed,
- $j \geq i$ is the index of the first symbol in T **not** being removed, and finally
- S is the string being inserted into T before index j.

Therefore, the definition of an edit is:

$$\texttt{edit}(T, i, j, S) = t_{0,i-1} \mid\mid S \mid\mid t_{j,n-1} \tag{6.1}$$

where n was the original length of T (prior to the edit being applied).

Note the following cases:

- If $i = j$, the edit operation performs an *insertion* of S between t_{i-1} and t_i.
- If $S = \emptyset$, the edit operation performs a *deletion* of symbols $t_{i,j-1}$ from T.
- We do not consider the case where both $i = j$ and $S = \emptyset$, as that operation leaves T unchanged.
- All other cases (where $j > i$ and $S \neq \emptyset$) are string *replacements*, where symbols $t_{i,j-1}$ are replaced by the symbols in S.

For the remainder of this thesis, if T represents the raw string prior to an edit, then T' represents the edited raw string. Similarly, if Π represents the compressed form of T, then Π' represents the compressed form of T'.

Note that the syntax of the `edit()` operation applies both to an uncompressed string and its compressed representation. We distinguish these cases as follows:

- A *raw edit* is an edit applied to a string T in its uncompressed form. This is denoted $\texttt{edit}(T, i, j, S)$.
- A *compressed edit* applies to the compressed representation of a string T. If Π stores the compressed representation of T, then the compressed edit is denoted $\texttt{edit}(\Pi, i, j, S)$. Note that the index values i and j refer to indices in the raw representation of the string T, and that S is stored in raw form.

Remark 6.1. Consider the case where we compress T' (i.e. the string T after applying a raw edit). This will produce a compressed representation of T' which we call Π'. This Π' does not necessarily consist of the same phrases as the parsing produced by applying the same edit to the compressed form Π of T.

To distinguish these cases, we refer to the compressed raw edit as Π', and the parsing following a compressed edit as Π'_e.

Formally,

$$\Pi' = \texttt{LZ-End}(\texttt{edit}(T, i, j, S)) \tag{6.2}$$

$$\Pi'_e = \texttt{edit}(\Pi, i, j, S) \tag{6.3}$$

Π' and Π'_e will both decode to the same string T', but the phrases in each parsing will not necessarily be the same.

6.1.3 Challenges of editing LZ-compressed data

Recall the LZ-End parsing of the string $T = \texttt{abracadabra}$:

$$\begin{aligned}\Pi\text{: } \pi_0 &= \langle 0, 0, \texttt{a}\rangle\\ \pi_1 &= \langle 0, 0, \texttt{b}\rangle\\ \pi_2 &= \langle 0, 0, \texttt{r}\rangle\\ \pi_3 &= \langle 0, 1, \texttt{c}\rangle\\ \pi_4 &= \langle 0, 1, \texttt{d}\rangle\\ \pi_5 &= \langle 2, 3, \texttt{a}\rangle\end{aligned}$$

Let us say we wish to perform the operation $\texttt{edit}(T, 1, 2, \texttt{d})$, so that $T = \texttt{abracadabra}$ becomes $T' = \texttt{adracadabra}$.

At first glance, it may appear that all we need to do is locate the phrase which encodes t_1 (i.e., phrase π_1), and change its innovation symbol from a '$\texttt{b}$' to a '$\texttt{d}$'. Doing that, we get:

$$\begin{aligned}\Pi'_e\text{: } \pi_0 &= \langle 0, 0, \texttt{a}\rangle\\ \pi_1 &= \langle 0, 0, \texttt{d}\rangle\\ \pi_2 &= \langle 0, 0, \texttt{r}\rangle\\ \pi_3 &= \langle 0, 1, \texttt{c}\rangle\end{aligned}$$

$$\pi_4 = \langle 0, 1, \mathtt{d} \rangle$$
$$\pi_5 = \langle 2, 3, \mathtt{a} \rangle$$

However, if we decode this, we get $T'' = \mathtt{adracadadra}$ – we see that the symbol t_1 has been edited correctly, but t_8 has also changed from a '`b`' to a '`d`'! This is because the phrase π_5 back-referenced phrases $\pi_{0,2}$. Changing any one of the phrases $\pi_{0,2}$ will therefore also affect phrase π_5.

This demonstrates the challenge of editing a symbol represented by the innovation symbol component of a phrase. Another challenge to overcome is: How to edit a symbol encoded by the back-reference component of a phrase? The editing algorithm presented in this chapter is the first algorithm to overcome both of these challenges.

6.2 Editing in LZ-End

This section explains each component of the editing algorithm for LZ-End and then shows how to tie the components together.

Let us consider the string

$$T = \mathtt{yzyyzzyyyzzzyyyyzzzzyyyyzzzzz},$$

and let us apply to it $\mathtt{edit}(T, 8, 10, \mathtt{zy})$. That is, we wish to change T to

$$T' = \mathtt{yzyyzzyyyzyzzyyyyzzzzyyyyzzzzz}$$

Note that the LZ-End parsing Π of T is:

$$\Pi\text{: } \pi_0 = \langle 0, 0, \mathtt{y} \rangle$$
$$\pi_1 = \langle 0, 0, \mathtt{z} \rangle$$
$$\pi_2 = \langle 0, 1, \mathtt{y} \rangle$$
$$\pi_3 = \langle 1, 1, \mathtt{z} \rangle$$
$$\pi_4 = \langle 2, 2, \mathtt{y} \rangle$$
$$\pi_5 = \langle 3, 2, \mathtt{z} \rangle$$
$$\pi_6 = \langle 4, 3, \mathtt{y} \rangle$$
$$\pi_7 = \langle 5, 3, \mathtt{z} \rangle$$
$$\pi_8 = \langle 7, 8, \mathtt{z} \rangle$$

The following sections apply each step of the editing algorithm to this example, before tying all steps together for the complete algorithm.

6.2.1 Identifying phrases to edit

The first step of the $\mathtt{edit}(\Pi, i, j, S)$ operation is to identify which phrase(s) encode the symbols we wish to edit. The `rank` operation makes this step very

easy. We call the phrases which encode the symbols we wish to edit *target* phrases:

$$a \leftarrow \texttt{rank}(i) \tag{6.4}$$

$$b \leftarrow \texttt{rank}(j) \tag{6.5}$$

In our example edit operation, $a = 4$ and $b = 5$.

Note that because we have $i \leq j$, then we must have $a \leq b$.

At this stage, we know that the edit operation is changing at least one symbol in π_a, and at least one symbol in π_b. We do not yet know if all symbols encoded by π_a and π_b are being edited. It is possible that the edit does not modify the prefix of π_a, or the suffix of π_b. We refer to these symbols as the *prefix* and *suffix* of the target phrases. The lengths of the prefix and suffix are:

$$pref\_len \leftarrow \pi_a.l - (\texttt{select}(a) - i) \tag{6.6}$$

$$suf\_len \leftarrow \texttt{select}(b) - j + 1 \tag{6.7}$$

Recall that while `rank` maps the index of a symbol in T to a phrase in Π, `select` acts as the inverse: mapping the index of a phrase in Π to the index of the last symbol in T which is encoded by the phrase.

In our example string, $pref\_len = 2$ and $suf\_len = 2$. These prefix and suffix lengths will be important in a later step.

6.2.2 Edit the target phrases

We can now edit the target phrases identified in the previous step. This involves deleting the target phrases, and encoding the inserted string S. Before deleting the target phrases, we first want to make sure we do not lose the prefix and suffix identified in the previous step.

In order to keep the symbols of the prefix of our target, we simply decode (using the `Random access` function Algorithm 3) the prefix of the first target phrase π_a, and prepend this string to our inserted string S, to be handled next. We do the same with the suffix, appending the decoded symbols to S. Formally,

$$S \leftarrow \texttt{Random access}(\Pi, i - pref\_len, pref\_len) \mid\mid S \mid\mid \texttt{Random access}(\Pi, j, suf\_len)$$

For our example, we decode the first two symbols of π_4 and the last two symbols

of π_5 and prepend or append them to S:

$$S \leftarrow \texttt{yyzyzz} \tag{6.8}$$

S is still in raw form, and must be parsed. We will show in the next chapter a method which uses the parsing of T to parse S. However, in this proof-of-concept algorithm, we parse S independently of T:

$$\Phi \leftarrow \texttt{LZ-End}(S)$$

In our example, parsing $S = \texttt{yyzyzz}$, we get:

$$\begin{aligned}\Phi\text{: } \phi_0 &= \langle 0, 0, \texttt{y}\rangle \\ \phi_1 &= \langle 0, 1, \texttt{z}\rangle \\ \phi_2 &= \langle 1, 2, \texttt{z}\rangle\end{aligned}$$

At a future stage, we will delete the target phrases π_a and π_b and insert Φ (the parsing of our inserted string S) in their place:

$$\Pi \leftarrow \pi_{0,a-1} \mid\mid \Phi \mid\mid \pi_{b+1,m-1}$$

where there were m phrases in the parsing Π prior to the edit operation.

The back-references of the phrases in Φ will be incorrect (if we use absolute phrase indices as the back-reference) – but this is easily corrected by adding a to each back-reference in Φ.

In our example, the parsing will then become:

$$\begin{aligned}\Pi\text{: } \pi_0 &= \langle 0, 0, \texttt{y}\rangle \\ \pi_1 &= \langle 0, 0, \texttt{z}\rangle \\ \pi_2 &= \langle 0, 1, \texttt{y}\rangle \\ \pi_3 &= \langle 1, 1, \texttt{z}\rangle \\ \cancel{\pi_4 = \langle 2, 2, \texttt{y}\rangle} \\ \cancel{\pi_5 = \langle 3, 2, \texttt{z}\rangle} \\ \pi_6 &= \langle 4, 3, \texttt{y}\rangle \\ \pi_7 &= \langle 5, 3, \texttt{z}\rangle \\ \pi_8 &= \langle 7, 8, \texttt{z}\rangle\end{aligned} \qquad \begin{aligned}\Phi\text{: } \phi_0 &= \langle 4, 0, \texttt{y}\rangle \\ \phi_1 &= \langle 4, 1, \texttt{z}\rangle \\ \phi_2 &= \langle 5, 2, \texttt{z}\rangle\end{aligned}$$

This shows what the parsing will become – that is, we will delete phrases π_4 and π_5 and insert Φ in their place. We do not perform this just yet, because it will be easier to do so at a later stage.

6.2.3 Identifying dependent phrases

We cannot simply replace the target phrases with their new encoding Φ: the result will generally not decode correctly. This is because some phrases (in

our example, phrases π_6, π_7 and π_8) either directly or indirectly reference the phrases we have edited. We say that such phrases *depend* on the target phrases.

An important distinction to make is the *order* of the dependent phrases: we say that phrases which back-reference our target phrases are *first-order* dependents. All other dependent phrases of first-order dependents are called *higher-order* dependents. Changing the parsing of a phrase will alter the decoding of all dependent phrases (first-order and higher-order dependents). We therefore need to identify these dependent phrases and "decouple" their back-references from the target phrases. We call this "mending" the parsing.

Remark 6.2. We only need to mend the parsing of the first-order dependent phrases. This is because the higher-order dependents (by definition) reference the first-order dependents. As long as the first-order dependents decode correctly, all higher-order dependents will also decode correctly.

In our example, π_6 and π_7 are first-order dependents and π_8 is a second-order dependent (itself being a first-order dependent of both π_6 and π_7). Our example will show how fixing the parsing of π_6 and π_7 alone is sufficient to fix the parsing of π_8.

Algorithm 4 identifies the first-order dependent phrases. The algorithm returns the index in Π of all first-order dependent phrases – these are stored in the array D.

Algorithm 4 `identify dependent phrases`(Π, a, b, suf_len)

Require: An array Π of LZ-End phrases
An integer a denoting the first target phrase
An integer b denoting the last target phrase
An integer suf_len denoting the number of symbols in b target phrase which are not part of the target.
Result: an array of phrase pointers, D.

1: $D \leftarrow \emptyset$
2: $X \leftarrow \{suf\_len\}$
3: $ptr \leftarrow b + 1$
4: **while** $ptr < \Pi.size$ **do**
5: $\quad X \leftarrow X \mid\mid x_{ptr-b-1} + \pi_{ptr}.l + 1$
6: $\quad$ **if** $\pi_{ptr}.b < a$ **then**
7: $\quad\quad$ **continue**
8: $\quad$ **else if** $\pi_{ptr}.b < b$ **or** $x_{\pi_{ptr}.b-b} < \pi_{ptr}.l$ **then**
9: $\quad\quad D \leftarrow D \mid\mid ptr$
10: $\quad$ **end if**
11: $\quad ptr \leftarrow ptr + 1$
12: **end while**
13: **return** D

The algorithm examines the back-reference of each phrase which follows b. If a phrase's back-reference is less than a, it cannot possibly be a dependent phrase (Line 6). If the back-reference is between a and b, the phrase is obviously dependent. This case is covered by the first part of the conditional statement at Line 8.

There is one more case: when the back-reference is greater than or equal to b. In this case, the phrase may yet be a dependent. Recall that a phrase references the l symbols which precede the end of the back-referenced phrase. For each phrase π_{ptr} whose back-reference is greater than b, we use the phrase's length, $\pi_{ptr}.l$, to determine whether the back-reference extends to the target phrase. Since this calculation will need to be performed for every single phrase which follows the target, we use a cache to accelerate this step. This cache is stored in the list X (Line 2).

For each phrase π_{ptr}, we compute the distance (in symbols) between that phrase and the last symbol of the target (Line 5). Using the distance of our previous phrase, this is a fast computation. The cache is initialised with $x_0 = suf\_len$ – this is because a phrase may back-reference the last phrase of our target without actually being a dependent on the target. Recall that the last suf_len symbols of π_b are not actually part of the target.

We use the cache X to quickly determine the distance (in symbols) of each phrase's back-reference from the target, and determine whether the phrase depends on the target by comparing the phrase length to this distance. The second part of Line 8 performs this calculation. If a phrase is dependent, we append its index to the array D which stores the indices of all dependent phrases.

Once all phrases after π_b have been processed, the algorithm returns the array of dependent phrases, D.

Applying this algorithm to our example identifies π_6 and π_7 as phrases that depend on the target phrases, but not π_8, as it is a second-order dependent.

6.2.4 Mending the parsing of dependent phrases

Each of the dependent phrases identified thus far in the editing process needs to be replaced with phrases which decode to the same raw string, but do not reference the target phrases.

This is achieved by replicating all phrases which are back-referenced (adjusting the lengths as necessary), and then writing out the innovation symbol component of the dependent phrase as its own separate phrase.

Let us demonstrate this with our example parsing, which is repeated below for reference:

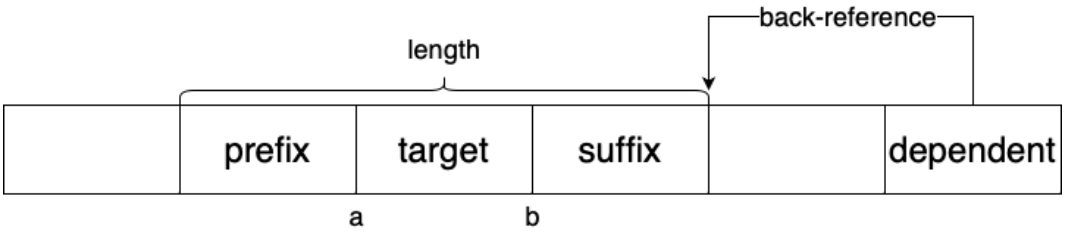


Figure 6.1: The possible components of a dependent phrase's back-reference. The dependent phrase may reference one or more prefix phrases which precede the target (labelled 'prefix'), one or more phrases in the target, and one or more suffix phrases which follow the target (labelled 'suffix').

$$\begin{aligned}
\Pi\text{: } \pi_0 &= \langle 0, 0, \mathtt{y} \rangle \\
\pi_1 &= \langle 0, 0, \mathtt{z} \rangle \\
\pi_2 &= \langle 0, 1, \mathtt{y} \rangle \\
\pi_3 &= \langle 1, 1, \mathtt{z} \rangle \\
\pi_4 &= \langle 2, 2, \mathtt{y} \rangle \\
\pi_5 &= \langle 3, 2, \mathtt{z} \rangle \\
\pi_6 &= \langle 4, 3, \mathtt{y} \rangle \\
\pi_7 &= \langle 5, 3, \mathtt{z} \rangle \\
\pi_8 &= \langle 7, 8, \mathtt{z} \rangle
\end{aligned}$$

The dependent phrase π_6 references the first target phrase π_4. This reference is exact, in that π_6 references all symbols in π_4, and no symbols from π_3. We can therefore replace π_6 with an exact copy of π_4. This does not encode the innovation symbol of π_6, so we need to encode this as a new phrase. Therefore, π_6 would be replaced by two phrases:

$$\begin{aligned}
\mu_0 &= \langle 2, 2, \mathtt{y} \rangle \\
\mu_1 &= \langle 0, 0, \mathtt{y} \rangle
\end{aligned}$$

Similarly, the dependent phrase π_7 would be replaced by an exact copy of the referenced target phrase π_5 and a single phrase which encodes the innovation symbol component of π_7:

$$\begin{aligned}
\nu_0 &= \langle 3, 2, \mathtt{z} \rangle \\
\nu_1 &= \langle 0, 0, \mathtt{z} \rangle
\end{aligned}$$

In these examples, each back-reference mapped to only one phrase. However, we can generalise the approach to cases where the back-reference points to multiple phrases. A dependent phrase can have three components to its back-reference, illustrated in Figure 6.1. The components consist of the phrases which precede the target (the prefix), the target phrases themselves, and any phrases which follow the target (the suffix). As our example string demonstrated, the prefix and suffix may not exist.

Each of these three components must be handled individually:

- **the *prefix* component:** If a back-reference points to multiple phrases

in this component, they can be amalgamated into a single back-reference. This is done be setting the replacement phrase to back-reference the final phrase in the prefix component, and setting the phrase length to the length of the symbols which make up the prefix component of the back-reference. The innovation symbol of this new phrase is the first symbol of the first phrase in the target.

- **the target phrases:** As we illustrated above, the replacement of the dependent phrase directly replicates each back-referenced phrase from the target. If we have a non-null prefix to the back-reference in the previous bullet point, we have to decrement the length of the first target phrase by 1 (to account for this symbol being the innovation symbol of the replacement phrase).
- **the *suffix* component:** This is the easiest case – we simply keep the original dependent phrase, and shorten its length such that it is no longer pointing at the target. This will now reference all phrases in the suffix, and handle the dependent phrase's original innovation symbol.

We formalise this process in Algorithm 5.

The algorithm has three parts, which correspond to the three components of a dependent phrase's back-reference: Lines 7–13 handle the prefix, Lines 14–20 handle the target phrases themselves, and Line 22 handles the suffix component. We will discuss each of these parts of the algorithm below.

Initialisation

The first two steps are initialising variables used in the remainder of the algorithm.

The first variable to be initialised is the array Φ. This array has one element for each dependent phrase D. Each element in Φ is itself an array of one or more phrases which replace the dependent phrase in D.

The variable k points to the index in D of the dependent phrase which is currently being processed.

For each dependent phrase which this algorithm processes, the variables a and b store the start and end (respectively) index positions of the string which is back-referenced by the dependent phrase (Lines 4 and 5). That is, the current dependent phrase will back-reference the string $t_{a,b}$.

At any given step of the algorithm, *len* will store the length of the string $t_{a,b}$ which has not yet been processed.

Algorithm 5 `re-parse dependent phrases`(Π, i, j, D, t)

Require: An array of LZ-End phrases Π.
An index i of the first symbol being removed.
An index j of the first symbol after i not being removed.
An array, D, referencing phrases in Π which depend on one or more symbols in the raw string $t_{i,j-1}$.
The raw symbol t, being the first target symbol of the edit operation. (i.e., t_i, prior to the edit being applied).
Result: An array, Φ, where each element corresponds to an element in D. Each element is itself an array of phrases which collectively replace the corresponding element in D.

1: $\Phi \leftarrow \{\{\emptyset\}^{D.size}\}$ i.e, an array of $D.size$ (initially empty) arrays of phrases.
2: $k \leftarrow 0$
3: **while** $k < D.size$ **do**
4: $b \leftarrow \texttt{select}(\pi_{d_k}.b)$
5: $a \leftarrow b - \pi_{d_k}.l + 1$
6: $len \leftarrow \pi_{d_k}.l$
7: **if** $a < i$ **then**
8: $\Phi_k \leftarrow \{\langle \texttt{rank}(i) - 1, i - a, t\rangle\}$
9: $len \leftarrow len - i - a - 1$
10: $ptr \leftarrow \texttt{rank}(i)$
11: **else**
12: $ptr \leftarrow \texttt{rank}(a)$
13: **end if**
14: **while** $len > 0$ **and** $ptr \leq \texttt{rank}(j)$ **do**
15: $tmp \leftarrow \pi_{ptr}$
16: $tmp.l \leftarrow \min(len - 1,\ tmp.l)$
17: $\Phi_k \leftarrow \Phi_k \mid\mid tmp$
18: $len \leftarrow len - tmp.l - 1$
19: $ptr \leftarrow ptr + 1$
20: **end while**
21: **if** $len > 0$ **then**
22: $\Phi_k \leftarrow \Phi_k \mid\mid \langle \pi_{d_k}.b, len, \pi_{d_k}.i\rangle$
23: **else**
24: $\Phi_k \leftarrow \Phi_k \mid\mid \langle 0, 0, \pi_{d_k}.i\rangle$
25: **end if**
26: $k \leftarrow k + 1$
27: **end while**
28: **return** Φ

The prefix component

Recall that the prefix component of a dependent phrase does not exist when $a \geq i$. In cases where the prefix does exist (i.e., $a < i$), we replace its encoding with that of a single phrase. This replacement phrase points to the last phrase of the prefix. The innovation symbol of the replacement phrase is therefore the first symbol of the first target phrase. This symbol is passed as a parameter to the

algorithm, and will have been accessed by a previous call to the `LZ-End random access`() function (Algorithm 3). This replacement phrase is constructed in Line 8, and becomes the first in our array of replacement phrases, Φ_k.

Remark 6.3. We denote an individual element of Φ with a capital letter (e.g., Φ_k) – this is because each element of Φ is itself an array.

We then need to decrement *len*, so that we account for the fact that we have replaced the encodings of the symbols in the prefix component of the back-reference.

Finally, we set *ptr* to point to the first phrase in our target whose encoding still needs to be replaced. This happens whether there is a prefix component to replace or not (note the **else** condition on Line 11).

The target phrases

We replace each target phrase simply by copying its encoding from our target parsing (see Line 15). There is a possibility that the back-reference of the dependent phrase may not be as long as the length of the phrase we have just copied. This can only happen for the first phrase of the target, and only if there was no prefix. Line 16 accounts for this by correcting the length of the phrase *tmp*.

After the length has (potentially) been altered, we append the copied phrase to the replacements array, Φ_k (Line 17), and account for the length of the symbols whose encodings we have replaced (Line 18).

The suffix component

We replace the suffix component of a dependent phrase (if it exists) with a single replacement phrase. This is simple, because the dependent phrase already points to this suffix. The only component of this phrase which must be changed is the length, which must be shortened so that the phrase does not reference the target phrases (Line 22).

If the suffix did not exist (i.e., *len* was zero at Line 21), we still need to replace the innovation symbol of the dependent phrase (Line 24), since this will not have been included in the direct copying of the target phrases.

Applying this to our example

Applying this to our example, we established above that the dependent phrase π_6 would be replaced by two phrases:

$$\mu_0 = \langle 2, 2, \mathrm{y} \rangle$$

$$\mu_1 = \langle 0, 0, \mathtt{y} \rangle$$

and that the dependent phrase π_7 would be replaced by:

$$\nu_0 = \langle 3, 2, \mathtt{z} \rangle$$
$$\nu_1 = \langle 0, 0, \mathtt{z} \rangle$$

Putting this all together, we have:

Π:	$\pi_0 = \langle 0, 0, \mathtt{y} \rangle$	Φ:	$\phi_0 = \langle 4, 0, \mathtt{y} \rangle$
	$\pi_1 = \langle 0, 0, \mathtt{z} \rangle$		$\phi_1 = \langle 4, 1, \mathtt{z} \rangle$
	$\pi_2 = \langle 0, 1, \mathtt{y} \rangle$		$\phi_2 = \langle 5, 2, \mathtt{z} \rangle$
	$\pi_3 = \langle 1, 1, \mathtt{z} \rangle$		
	~~$\pi_4 = \langle 2, 2, \mathtt{y} \rangle$~~	μ:	$\mu_0 = \langle 2, 2, \mathtt{y} \rangle$
	~~$\pi_5 = \langle 3, 2, \mathtt{z} \rangle$~~		$\mu_1 = \langle 0, 0, \mathtt{y} \rangle$
	~~$\pi_6 = \langle 4, 3, \mathtt{y} \rangle$~~		
	~~$\pi_7 = \langle 5, 3, \mathtt{z} \rangle$~~	ν:	$\nu_0 = \langle 3, 2, \mathtt{z} \rangle$
	$\pi_8 = \langle 7, 8, \mathtt{z} \rangle$		$\nu_1 = \langle 0, 0, \mathtt{z} \rangle$

We need to insert the phrases Φ in place of $\pi_{4,5}$, μ in place of π_6 and ν in place of π_7. Unfortunately, doing so will introduce an error: π_8 will not decode correctly. This is because we have changed the number of phrases in the parsing, so that the back-references of the original phrases will need to be updated. The next section addresses this issue.

6.2.5 Adjust back-references

The number of phrases used to encode the target and dependent symbols has likely changed in the previous step. This may affect the decoding of all phrases which follow the target, even of those phrases which are not dependents. We therefore need to correct the back-references of phrases after the target, to ensure that they back-reference the correct phrases.

The algorithm to make these corrections therefore needs to consider:

- the location and number of target phrases removed,
- the number of phrases (if any) inserted in place of the target,
- the position of each dependent phrase, and
- the number of phrases replacing each dependent phrase.

Algorithm 6 describes a method to fix the back-references. The algorithm processes each phrase after the final target phrase.

Phrases whose back-references point to phrases before the target do not need adjusting, hence the condition at Line 2. If the phrase's back-reference points to a phrase after the final target phrase, we increment its back-reference by the difference between the number of phrases in the original target and the number of phrases inserted in their place (Line 3).

Algorithm 6 `adjust pointers`(Π, a, l, z, D, Φ)

Require: An array Π of LZ-End phrases.
A pointer a to the first phrase whose index may need adjusting.
The number of phrases l which were removed from Π.
The number of phrases z in the encoding of the inserted string S.
An array D of pointers to the dependent phrases.
An array Φ of replacement phrases for each dependent.
Result: The array Π, once its index pointers have been corrected to account for the modification.

```
1: while a < Π.size do
2:     if π_a.b ≥ x then
3:         tmp ← π_a.b + z − l
4:         x ← 0
5:         while x < D.size do
6:             if π_a.b ≥ d_x then
7:                 tmp ← tmp + Φ_x.size − 1
8:             else
9:                 break
10:            end if
11:            x ← x + 1
12:        end while
13:        π.b ← tmp
14:    end if
15:    a ← a + 1
16: end while
17: return Π
```

Remark 6.4. Note that in Algorithm 6, the parsing Π has not yet been edited: neither the target phrases, nor the dependent phrases, have been replaced yet. This is why the algorithm compares each phrase's back-reference to the dependents array D.

The algorithm then compares the back-reference against each of the dependents. As long as the back-reference exceeds the index of the dependent phrase, we need to increment the back-reference by the difference introduced by replacing the dependent phrase (Line 7). As soon as a dependent phrase has an index greater than the back-reference, we can stop processing the phrase, hence the early exit condition (Line 9).

Recall that our parsing (after replacing the target and dependent phrases) will become:

$$\begin{aligned}\Pi\colon\ \pi_0 &= \langle 0, 0, \mathtt{y}\rangle\\ \pi_1 &= \langle 0, 0, \mathtt{z}\rangle\\ \pi_2 &= \langle 0, 1, \mathtt{y}\rangle\\ \pi_3 &= \langle 1, 1, \mathtt{z}\rangle\end{aligned}$$

$$\phi_0 = \langle 4, 0, \mathtt{y} \rangle$$
$$\phi_1 = \langle 4, 1, \mathtt{z} \rangle$$
$$\phi_2 = \langle 5, 2, \mathtt{z} \rangle$$
$$\mu_0 = \langle 2, 2, \mathtt{y} \rangle$$
$$\mu_1 = \langle 0, 0, \mathtt{y} \rangle$$
$$\nu_0 = \langle 3, 2, \mathtt{z} \rangle$$
$$\nu_1 = \langle 0, 0, \mathtt{z} \rangle$$
$$\pi_8 = \langle 7, 8, \mathtt{z} \rangle$$

The only dependent phrase which needs to be edited here is the final phrase π_8, which becomes $\pi_8 = \langle 10, 8, \mathtt{z} \rangle$, pointing at the phrase ν_1. The entire parsing now decodes correctly, reflecting the edit that was made.

Remark 6.5. Note that while Algorithm 6 presents a quadratic-time method to adjust the pointers, this can be done in linear time. By storing a mapping between phrase indices in the original parsing Π and their new index positions in the edited parsing Π', one can do away with the inner `while` loop at Line 5.

6.2.6 Replace the dependent phrases

Algorithm 5 calculated the parsing to replace each dependent phrase, but it did not actually replace the dependent phrases in Π. Algorithm 7 below performs the actual replacement of those dependent phrases. Note that this algorithm is called *before* the target phrases have been replaced. This allowed us to use the pointers D in Π to identify each dependent phrase.

The algorithm replaces each dependent phrase, starting with the last dependent phrase, and moving to the first. This prevents previously-replaced

Algorithm 7 `replace dependent phrases`(Π, D, Φ):

Require: The LZ-End parsing Π consisting of m phrases, which is being edited.
An array D of pointers to phrases in Π to be edited.
An array Φ, where each element is an array of phrases to replace the dependent phrase referenced by the corresponding element of D.

1: $ptr \leftarrow D.size - 1$
2: **while** $ptr \geq 0$ **do**
3: $\quad \Pi \leftarrow \pi_{0,d_{ptr}-1} \mid\mid \Phi_{ptr} \mid\mid \pi_{d_{ptr}+1,m-1}$
4: $\quad m \leftarrow \Pi.size$
5: $\quad ptr \leftarrow ptr - 1$
6: **end while**
7: **return** Π

dependent phrases from affecting the pointers to the dependent phrases.

6.2.7 Putting it all together

The complete LZ-End editing algorithm is therefore given in Algorithm 8. The initial steps calculate the phrases which encode the symbols which are being removed, as well as the length of the prefix and suffix to not be removed from these phrases.

Line 6 prepends the prefix of phrase a and appends the suffix of phrase b to the inserted string S. The algorithm then identifies the dependent phrases and calculates their replacements (Lines 7 and 8).

Next, we parse the inserted string S. We then set the back-reference component of each phrase so that it decodes correctly when the parsing is inserted into Π (between Lines 9 and 12).

Finally, we adjust the pointers of the phrases which follow the target (Line 13), replace the dependent phrases (Line 14) and insert the parsing of string S into Π (Line 15).

Algorithm 8 `LZ-End edit`(Π, i, j, S):

Require: The LZ-End parsing Π consisting of m phrases, which forms the LZ-End parsing of the string T of length n.
Integers i, j, such that $0 \leq i \leq j$.
A string of symbols S.

Result: The array Π which has had symbols $t_{i,j-1}$ removed and S inserted in their place.

1: $a \leftarrow$ `rank`(i)
2: $b \leftarrow$ `rank`(j)
3: $pref\_len \leftarrow \pi_a.l - ($`select`$(a) - i)$
4: $suf\_len \leftarrow$ `select`$(b) - j + 1$
5: $t \leftarrow$ `random access`$(\Pi, i, 1)$
6: $S \leftarrow$ `random access`$(\Pi, i - pref\_len, pref\_len) \mid\mid S \mid\mid$ `random access`(Π, j, suf_len) (Algorithm 3)
7: $D \leftarrow$ `identify dependent phrases`(Π, a, b, suf_len) (Algorithm 4)
8: $\Phi \leftarrow$ `re-parse dependent phrases`(Π, i, j, D, t) (Algorithm 5)
9: $\Theta \leftarrow$ `LZ-End`(S) (Algorithm 2)
10: **for all** θ in Θ **do**
11: $\quad \theta.b \leftarrow \theta.b + a$
12: **end for**
13: $\Pi \leftarrow$ `adjust pointers`$(\Pi, b+1, b-a+1, \Theta.size, D, \Phi)$ (Algorithm 6)
14: $\Pi \leftarrow$ `replace dependents`(Π, D, Φ) (Algorithm 7)
15: $\Pi \leftarrow \pi_{0,a-1} \mid\mid \Theta \mid\mid \pi_{b+1,m}$
16: **return** Π

6.3 How edits affect the LZ-End parsing

Recall that one notable characteristic of the LZ-End parsing is *phrase uniqueness*, whereby no two phrases encode identical substrings. This is an important property which is used in the proof that the LZ-End parsing of a string is coarsely optimal [2].

The editing algorithm, however, breaks this property. When we replace a dependent phrase, we explicitly copy prior back-references (Algorithm 5), and we do this repeatedly for all dependent phrases. In addition, we parse the inserted string S in isolation from the string T – that is, the symbols of S are encoded without any back-references to phrases encoding T. In addition, no phrases in T back-reference the encoding of S.

Therefore, applying a single edit to the LZ-End parsing may result in a parsing which is not coarsely optimal. This is a theoretical result; in this thesis we empirically evaluate how an edit affects the parsing in practice. For the remainder of this chapter, we analyse the theoretical properties of the editing algorithm and discuss its shortcomings. In the next chapter, we present an improved and optimised editing algorithm. For conciseness, we empirically evaluate the algorithm presented in this chapter alongside the improved version of the next chapter (see Section 7.4.2).

Remark 6.6. The LZ-End edit function has to parse the inserted string S in isolation from the string T which is being edited: Using the prefix of T which precedes the edit location to parse the string S is not possible, as this requires knowledge of the decoded prefix of T. To decode the entire part of T which precedes the edit location would violate the conditions of random access/edits, which imposes a limit on the number of symbols to be accessed/edited (see Definition 2.11).

6.4 Time and memory requirements

This section will analyse the time and memory required to perform an edit. We analyse each of Algorithms 4 to 6 individually, and then summarise the time and memory requirements of the editing algorithm (Algorithm 8) as a whole.

Identifying dependent phrases

Each operation performed by Algorithm 4 is a constant-time addition, a comparison, or assignment. The algorithm iterates once for each phrase which comes after the final target phrase. The time is therefore bounded by $O(\Pi.size-$

b), where b is the location of the last target phrase (we use the variable names of Algorithm 8).

The memory requirements are $O(\Pi.size - b)$, since the maximum possible size of the dependents array D is $\Pi.size - b$.

Re-parsing the dependent phrases

Algorithm 5 iterates once per dependent phrase. In the worst-case, the cost of replacing each dependent phrase consists of:

- a `rank` operation
- a `select` operation
- $b - a$ iterations of the `while` loop at Line 14 (where π_a and π_b are the first and last target phrases).

Recall that the indexable dictionary used by the original LZ-End parsing is static. This means that the data structure supporting constant-time `rank` and `select` queries will need to be rebuilt after each edit operation. Alternatively, a dynamic structure supporting `rank`/`select` queries could be used; however, these dynamic structures may not facilitate constant-time queries. We discuss this further in Sections 6.5 and 7.4.1.2. For the remainder of this chapter, we leave the cost of the `rank` and `select` queries in the run-time functions in which they appear, so that it is clear how the choice of different indexable dictionaries will affect the run-time of the edit function.

The space required to store the replacement phrases for each dependent is similarly $O(b - a)$.

Adjusting pointers

As we point out in Remark 6.5, Algorithm 6 can be improved by calculating a one-to-one mapping between original phrase indices in Π, and the new indices in the edited parsing Π'. This can be done in time $\Theta(\Pi.size - b)$. Once this precomputation is complete, adjusting each phrase's back-reference is a single constant-time operation. Since there are $\Pi.size - b$ phrases to adjust, the runtime of this algorithm is $\Theta(\Pi.size - b)$.

Algorithm 6 uses $\Theta(\Pi.size - b)$ space.

Total cost of an LZ-End edit operation

The total time complexity of the `LZ-End edit()` function (Algorithm 8) is therefore:

$$O((j-i+\mathit{pref\_len}+\mathit{suf\_len})\times(\texttt{rank}+\texttt{select})+\Pi.\mathit{size}-b+ \\ (b-a)\times(\Pi.\mathit{size}-b+\texttt{rank}+\texttt{select})+\mathcal{F}(S.\mathit{size})+(\Pi.\mathit{size}-b))$$

which reduces to:

$$O((j-i+\bar{l})\times(\texttt{rank}+\texttt{select})+(b-a)\times(\Pi.\mathit{size}-b+\texttt{rank}+\texttt{select})+ \\ \mathcal{F}(S.\mathit{size})+(\Pi.\mathit{size}-b))$$

where $\mathcal{F}(S.\mathit{size})$ is the cost of parsing the inserted string S, and $\bar{l} > \mathit{pref\_len}$, $\mathit{suf\_len}$ is the maximum length of a phrase in Π, prior to the edit being applied.

Now, $j-i \geq b-a$, which means that we can further reduce this formula to:

$$O((j-i+\bar{l})\times(\texttt{rank}+\texttt{select})+(b-a)\times(\Pi.\mathit{size}-b)+ \\ \mathcal{F}(S.\mathit{size})+(\Pi.\mathit{size}-b))$$

Finally, the term $(b-a)\times(\Pi.\mathit{size}-b)$ dominates the $(\Pi.\mathit{size}-b)$ term:

$$O((j-i+\bar{l})\times(\texttt{rank}+\texttt{select})+(b-a)\times(\Pi.\mathit{size}-b)+\mathcal{F}(S.\mathit{size})) \quad (6.9)$$

This shows that the cost of an edit operation is mainly dependent on:

- the size of the edit ($j-i$, $b-a$, and the size of the inserted string S),
- the length of the longest phrase in Π, prior to the edit being applied ($\bar{l}$),
- the cost of the `rank` and `select` queries,
- the size of the parsing ($\Pi.\mathit{size}$), and
- the position of the edit ($\Pi.\mathit{size}-b$).

In the next chapter, we will show how to adapt the LZ-End parsing so that the size of the parsing and the position of the edit have no impact on the time complexity of the edit operation.

An LZ-End edit operation requires $O(\texttt{max}(\Pi.\mathit{size}, \Pi'.\mathit{size}))$ space. This is the benefit of performing a compressed edit for LZ-End: the space complexity of the edit is a function of the *compressed* size of the string, rather than the raw size of the string being edited.

6.5 A note on `rank` and `select` queries

The reader will have noticed that we have abstracted away the `rank` and `select` queries, as well as any data structure which supports these queries. This has been a deliberate decision, one which the editing algorithm has been deliberately designed to accommodate: Note that Algorithm 8 first changes the parsing Π at Lines 13 to 15. All `rank` and `select` queries are made prior to this change, which means that we do not require a dynamic structure to support these queries. Also note that the cost of both the `rank` and `select` operations appears in the cost formula of the `LZ-End edit()` function (Equation (6.9)).

This means that our method of editing LZ-End compressed data (Algorithm 8) is not dependent on a particular data structure or format of LZ-End parsing used to answer `rank` and `select` queries.

If any data structure is used to answer `rank` and `select` queries, this must be rebuilt after applying the edit algorithm. We do not include this in the cost of the editing algorithm, since this same method would need to be applied if we decompressed, edited and recompressed the string: In either case, we need to reconstruct whatever data structure is used to answer `rank` and `select` queries. This data structure could be the indexable dictionary used by the original LZ-End parsing ([2, 61]), or the pre-computed table mentioned earlier in this thesis (Section 4.5.1).

The indexable dictionary by Raman *et al.* is *static*, meaning that if we make any changes to it, we must decode the compressed bit-vector, edit the vector and re-construct the compressed bit-vector. *Dynamic* indexable dictionaries do exist (see for example [77, 78, 79, 80]); the downside is that these have non-constant run-times for `rank` and `select` queries, and the cost of editing the structure is also non-constant. We leave it as future work to determine whether or not it is beneficial to use these dynamic structures, or simply reconstruct the compressed bit-vector of Raman *et al.* with each edit.

6.6 Conclusion

This chapter presented the first algorithm to edit data in its LZ-compressed form. The cost of the editing algorithm depends largely on the position of the edit within the string – this gets particularly costly when editing near the start of the string. In the next chapter, we present a new LZ-type parsing, and show how the structure of this parsing supports edits with a much improved cost function.

Chapter 7

LZ-Local: introducing a sliding window into the LZ-End parsing

This chapter introduces a sliding window into the LZ-End parsing. The sliding window limits the number of dependents each phrase can have; this dramatically improves the performance of the editing algorithm from the previous chapter.

The chapter starts by defining the new parsing, formed by incorporating a sliding window into LZ-End: We call this parsing *LZ-Local*. We choose this name because the sliding window means that all of the random-access and random-edit properties of the LZ-End parsing become locally accessible/editable (as we shall see). Refer back to Section 2.6.3 for a distinction between random access/edits, and local access/edits. We go on to discuss the implications which the sliding window has on the compression ratio, and how the editing algorithm is affected by the sliding window. Finally, we empirically evaluate the editing algorithm against various window sizes.

This chapter is squarely focused on the structure of the LZ-Local parsing, and the implications of the sliding window on editing. Like the original LZ-77 paper ([40]), we do not present an algorithm to efficiently construct the parsing, but leave this as future work.

7.1 LZ-Local parsing

The sliding window which we introduce into LZ-End is exactly the same as the sliding window of the original LZ-77 parsing (see Section 3.2). That is, for a window size of $w > 0$, every symbol encoded by an LZ-Local phrase is at most w symbols away from the symbol it back-references.

Recall that the back-references in the LZ-End parsing refer to absolute phrase indices. We change this for LZ-Local by using relative phrase indices: That is, if phrase π_j has a back-reference of x, it is pointing to phrase π_{j-x}. This means that the sliding window places an upper bound on the number of

bits required to encode any back-reference ($\lceil \log w \rceil$).

Definition 7.1. The LZ-Local parsing of a string $T = t_0 t_1 \dots t_{n-1}$ is a sequence of $m \leq n$ phrases Π. If the first $q < m$ phrases of Π encode the first $i < n$ symbols of T, then the phrase π_q encodes the longest prefix of $t_{i,n-1}$ which is a suffix of the concatenation of phrases $\pi_{j,k}$ for some $p \leq j \leq k < q$, where $w > 0$ is a constant, $p = 0$ if $i < w$, otherwise p is the index in Π of the phrase encoding symbol t_{i-w}.

Example 7.1. Let us parse the string $T =$ `abracadabraracada` using the LZ-End and LZ-Local parsings (i.e., with and without a sliding window). In the case of LZ-Local, use a window size $w = 8$.

LZ-End (no window)	LZ-Local (window size = 8)
Π: $\pi_0 = \langle 0, 0, \mathtt{a} \rangle$	Φ: $\phi_0 = \langle 0, 0, \mathtt{a} \rangle$
$\pi_1 = \langle 0, 0, \mathtt{b} \rangle$	$\phi_1 = \langle 0, 0, \mathtt{b} \rangle$
$\pi_2 = \langle 0, 0, \mathtt{r} \rangle$	$\phi_2 = \langle 0, 0, \mathtt{r} \rangle$
$\pi_3 = \langle 0, 1, \mathtt{c} \rangle$	$\phi_3 = \langle 3, 1, \mathtt{c} \rangle$
$\pi_4 = \langle 0, 1, \mathtt{d} \rangle$	$\phi_4 = \langle 4, 1, \mathtt{d} \rangle$
$\pi_5 = \langle 2, 3, \mathtt{a} \rangle$	$\phi_5 = \langle 3, 3, \mathtt{a} \rangle$
$\pi_6 = \langle 4, 5, \mathtt{a} \rangle$	$\phi_6 = \langle 1, 2, \mathtt{c} \rangle$
	$\phi_7 = \langle 2, 1, \mathtt{d} \rangle$
	$\phi_8 = \langle 0, 0, \mathtt{a} \rangle$

Here, the two parsings are identical for phrases $\pi_{0,5} = \phi_{0,5}$: The only differences are the relative versus absolute back-references. The parsings diverge for the last six symbols, $t_{11,16}$: The original LZ-End parsing represents these symbols as a single phrase π_6. LZ-Local, however, represents these symbols as three phrases, $\phi_{6,8}$. This is due to the constraint that no phrase be more than $w = 8$ symbols from its back-reference.

7.1.1 Properties of the LZ-Local parsing

This section discusses how the differences between the LZ-End and LZ-Local parsings affect the relative performance of their compression and random access algorithms.

Coarse optimality

Recall that the proof of the LZ-End parsing's coarse optimality relied on each phrase in the parsing being unique (recall Definition 2.10, Section 6.3 and [2]). The introduction of a sliding window means that the LZ-Local phrases are not

unique; therefore, we have not been able to show that the LZ-Local parsing is coarsely optimal. Investigating this remains future work: a proof may involve using the upper bound on the number of bits used to encode the back-reference component of each LZ-Local phrase, in conjunction with Theorem 2.2.

Random access

The random-access property of LZ-End becomes *local access* in LZ-Local. Recall that an arbitrary phrase in the LZ-End can be retrieved in linear time. However, this requires random access to all phrases which precede the phrase we wish to decode.

LZ-Local allows the same linear-time decoding of an arbitrary phrase: the difference is that there is an upper bound on the distance between the phrase being decoded and the furthest phrase which we may need to access.

Theorem 7.1. LZ-Local allows local access to the symbols of an arbitrary phrase: that is, symbol t_i can be recovered from the parsing Π by reading only phrases encoding the symbols $t_{i-l\times w,i}$, where l is the length of the phrase encoding t_i and w is the window size used in the LZ-Local parsing.

Proof. Recall that Algorithm 3 (the `random access()` algorithm which applies to both LZ-End and LZ-Local) allows decoding a phrase π of length l in at most l steps [2]. The algorithm achieves this by recursively back-referencing phrases and reading their innovation symbols. We also know that Algorithm 3 makes a maximum of l recursive calls.

From the definition of the LZ-Local parsing, we know that each recursive call of Algorithm 3 will read the innovation symbol of a phrase at most w symbols preceding the previous phrase. □

Note that the actual random-access algorithm does not change – that is, Algorithm 3 applies to LZ-Local parsing, without alteration. However, the structure of the LZ-Local parsing optimises the "locality" of phrase decoding: randomly accessing the symbols of an LZ-End phrase π_x may require reading any phrase in $\pi_{0,x-1}$. Randomly accessing the symbols of an LZ-Local phrase π_x, however, only requires accessing phrases which encode symbols in $t_{i-l\times w,i}$, if π_x is of length l, and π_x encodes symbol t_i (Theorem 7.1).

Random editing

The sliding window of LZ-Local places an upper bound on the number of possible dependent phrases (and therefore phrases which need to be replaced). Section 7.3 discusses this in detail.

7.2 Evaluating the LZ-Local parsing

Before discussing the LZ-Local editing algorithm, it is important to examine the effect which the sliding window (and altering the size thereof) has on the compression ratio. We have discussed the theoretical implications already (namely, the loss of coarse optimality); here, we empirically evaluate the effect which varying the window size has on compressibility. This section applies the methodology introduced in Section 5.2, using the files from the Canterbury corpus [70] and strings of calibrated entropy [72]. This section measures the effect which varying window sizes have on the LZ-Local compression ratio, how the compression achieved by a given window size compares to LZ-77, and what window size is required for LZ-Local to become competitive with LZ-End.

Remark 7.1. Note that if the window size is larger than the string being compressed, the LZ-Local parsing will match the LZ-End parsing. Therefore, as the window size increases, we expect the LZ-Local compression ratio to approach that of LZ-End.

7.2.1 Canterbury corpus

We begin evaluating the LZ-Local algorithm on the Canterbury corpus, and present the results in Figures 7.1 to 7.18. The LZ-End algorithm has already been evaluated against the Canterbury corpus in [2]. The astute reader might notice that our results are close to, but do not match, those reported in [2]. We did manage to closely match the results in [2] by bit-packing the phrase back-references. However, for this evaluation we do not use bit-packing (recall Section 5.3).

The general patterns we observe in the Canterbury corpus files are:

- For the repetitive texts `aaa.txt` (Figure 7.1) and `alphabet.txt` (Figure 7.3), LZ-77 achieves very good compression for all window sizes. The LZ-Local compression rate, on the other hand, is highly dependent on the window size. For small window sizes, LZ-Local compresses the repetitive texts rather badly in comparison to LZ-77.
- For most non-repetitive files, changing the window size has a similar effect on LZ-77 compression as on LZ-Local. LZ-77 overall compresses better than LZ-Local. This is not surprising, given that the LZ-Local parsing is based on LZ-77, with added constraints.
- LZ-77 outperforms LZ-Local and LZ-End for most files. The notable exception to this is `kennedy.xls` (Figure 7.10).
- LZ-End usually outperforms LZ-Local for small window sizes, but LZ-Local converges on the LZ-End compression ratio as the window size gets

larger.

Remark 7.2. We omit the Canterbury file `a.txt`, as it is of no interest for our analysis: All LZ parsers will represent this file as a single phrase.

Remark 7.3. The LZ-End parsing does not have a sliding window. Therefore, in all graphs plotting compression ratio vs. window size, the LZ-End compression ratio is constant, and depicted with a horizontal line spanning all window sizes.

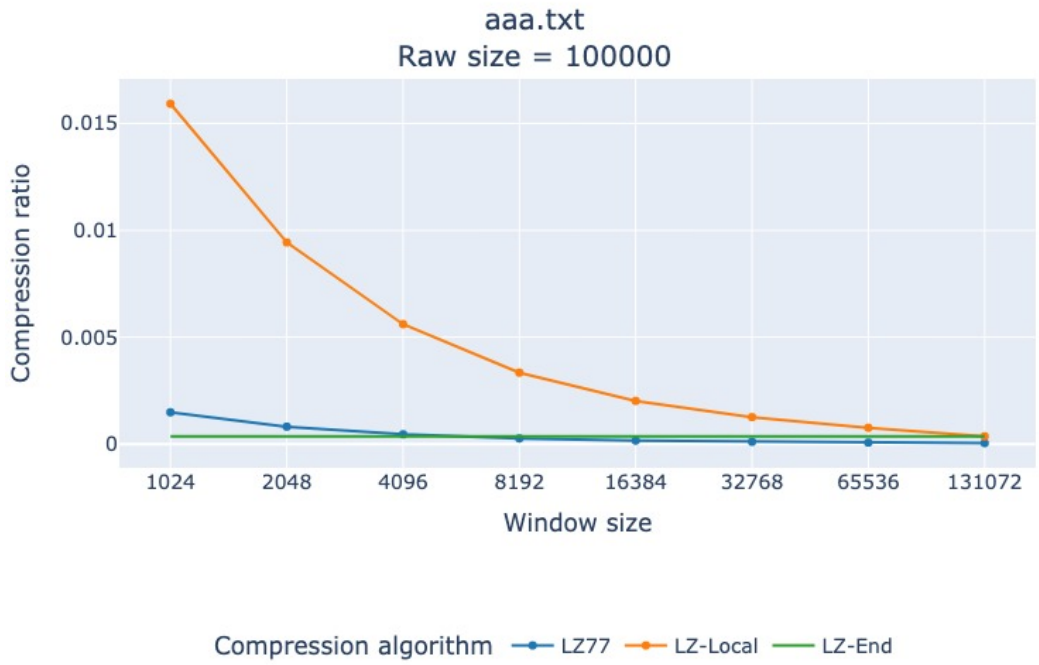


Figure 7.1: For this highly repetitive text, LZ-Local performs quite badly for small window sizes. As the window size increases, LZ-Local is slow to converge to the LZ-End compression ratio. The LZ-77 compression rate, on the other hand, is much less dependent on the window size.

Figure 7.2: For the first English text file, the size of the sliding window has a similar impact on LZ-Local compression as on LZ-77. However, LZ-77 is superior for all window sizes. Note that LZ-Local's performance converges on LZ-End, even for window sizes smaller than the overall text length.

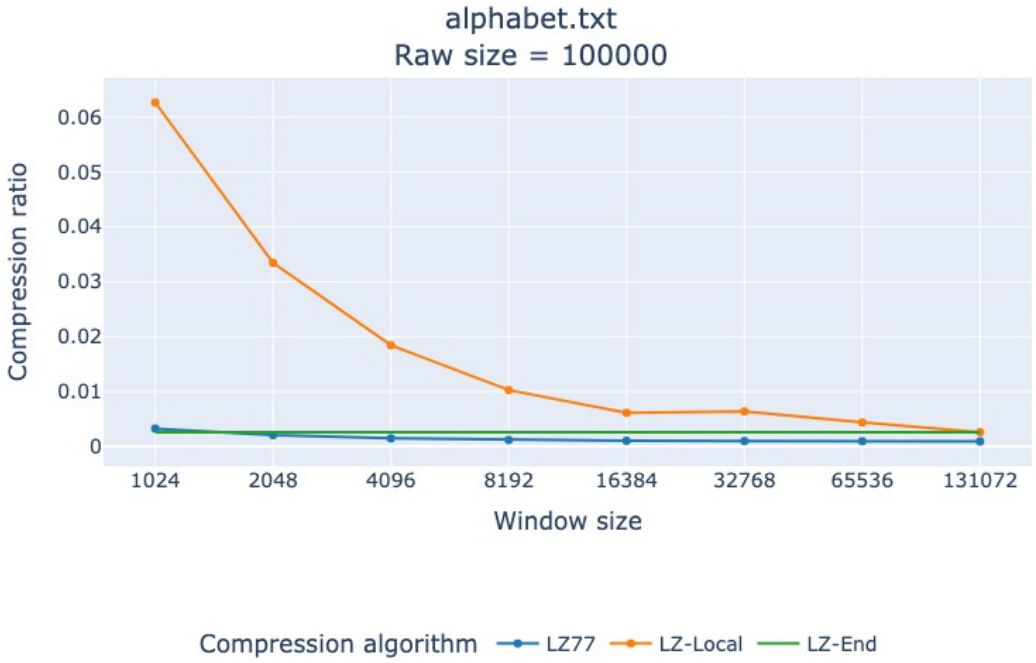


Figure 7.3: We start to see a pattern emerge with highly repetitive texts: Similar to Figure 7.1, LZ-77 compresses the text very well, even for small window sizes. LZ-Local, however, is slow to converge on the LZ-End compression rate as the window size increases.

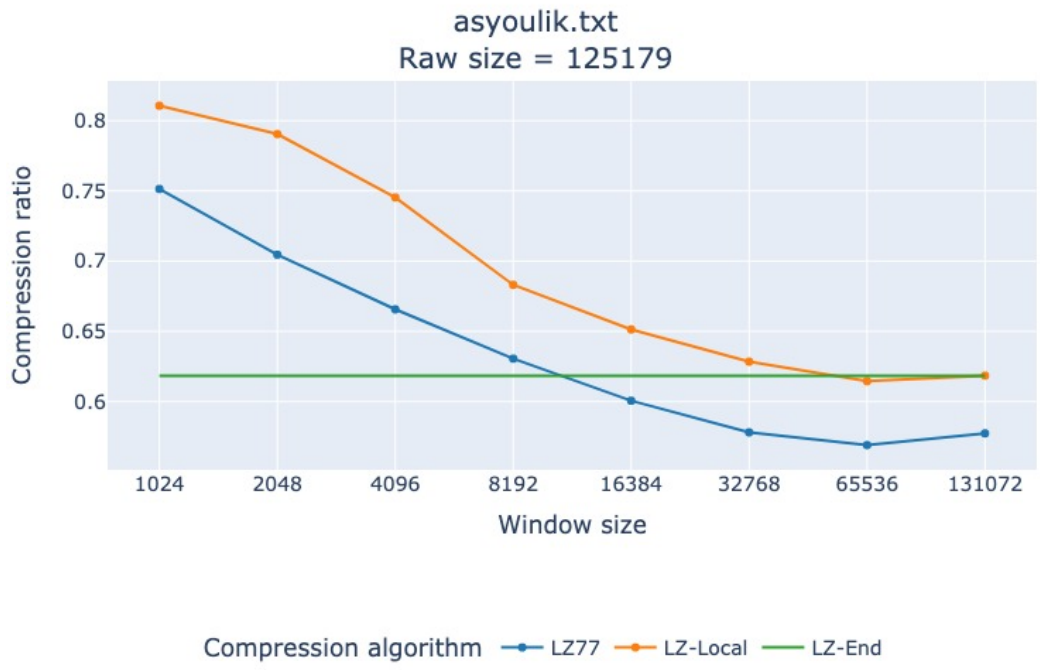


Figure 7.4: This text file is consistent with the general observations that varying the window size has a similar effect on LZ-Local compression as on LZ-77, with LZ-77 performing better overall. Again, LZ-Local compression rate converges on the LZ-End compression rate for window sizes less than the text length.

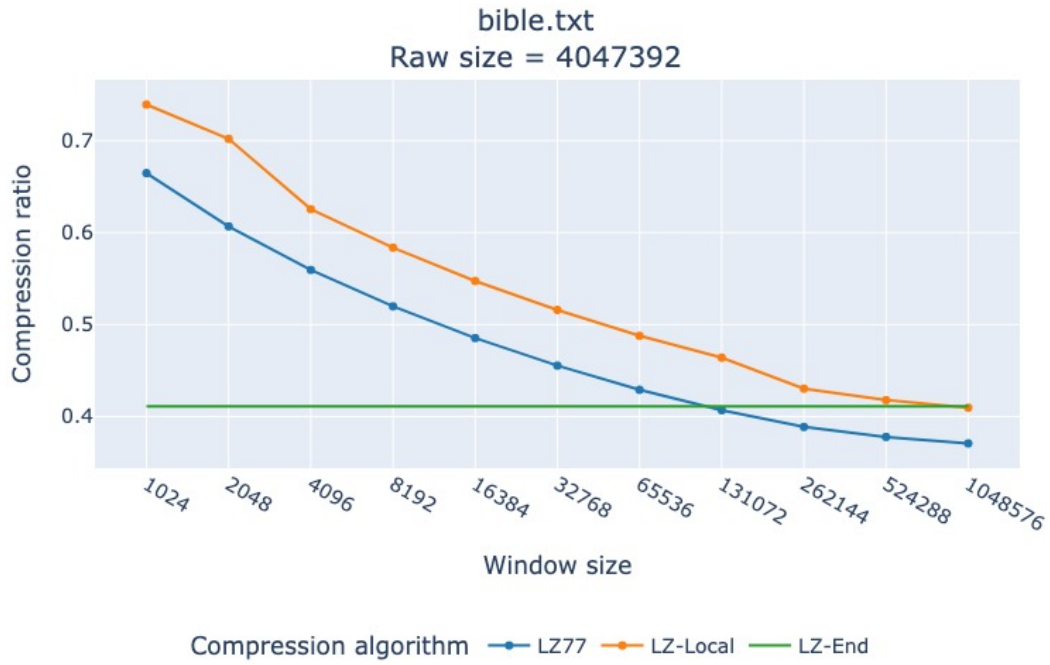


Figure 7.5: Our previous observation for English text files continue here: the window size has a similar effect on LZ-Local as on LZ-77, with LZ-77 performing better overall. The LZ-Local compression rate has converged on that of LZ-End, even though the file is approximately four times larger than the largest window size in this graph.

Figure 7.6: This file continues the trend we saw on English text files.

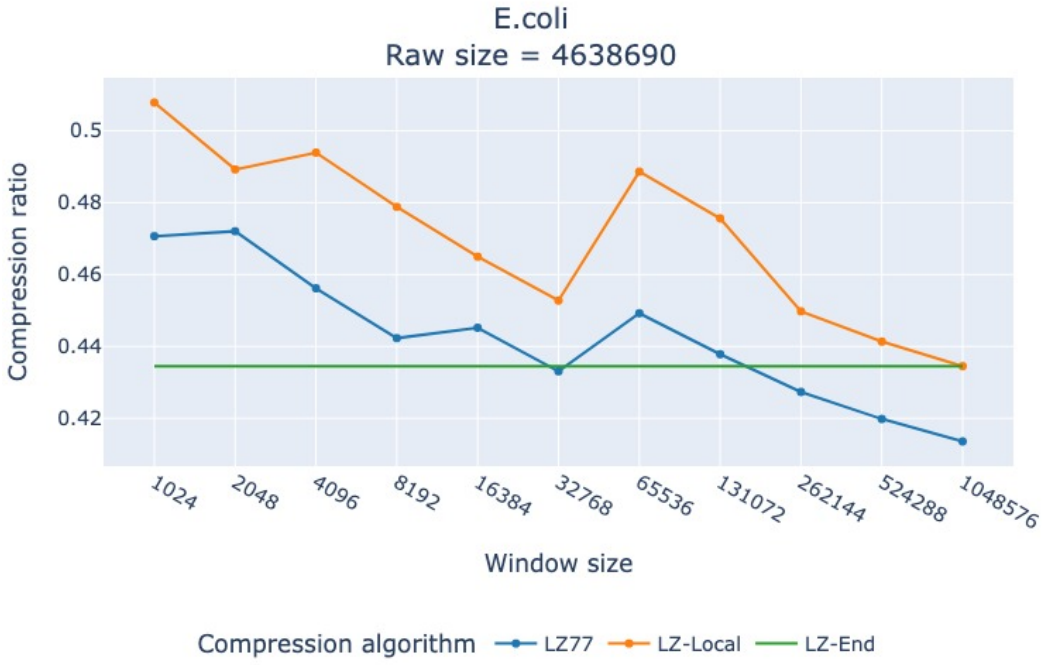


Figure 7.7: Note the same results as for previous medium-entropy (text) files. However, note the jump in compression ratio for both LZ-77 and LZ-Local, with window sizes between 32,768 and 65,536. This is because, for both parsings, the increased window size led to a dramatic increase in the size of each phrase, with minimal reduction in the number of phrases for either parsing.

Figure 7.8: This file continues the trend observed for medium-entropy, non-repetitive files.

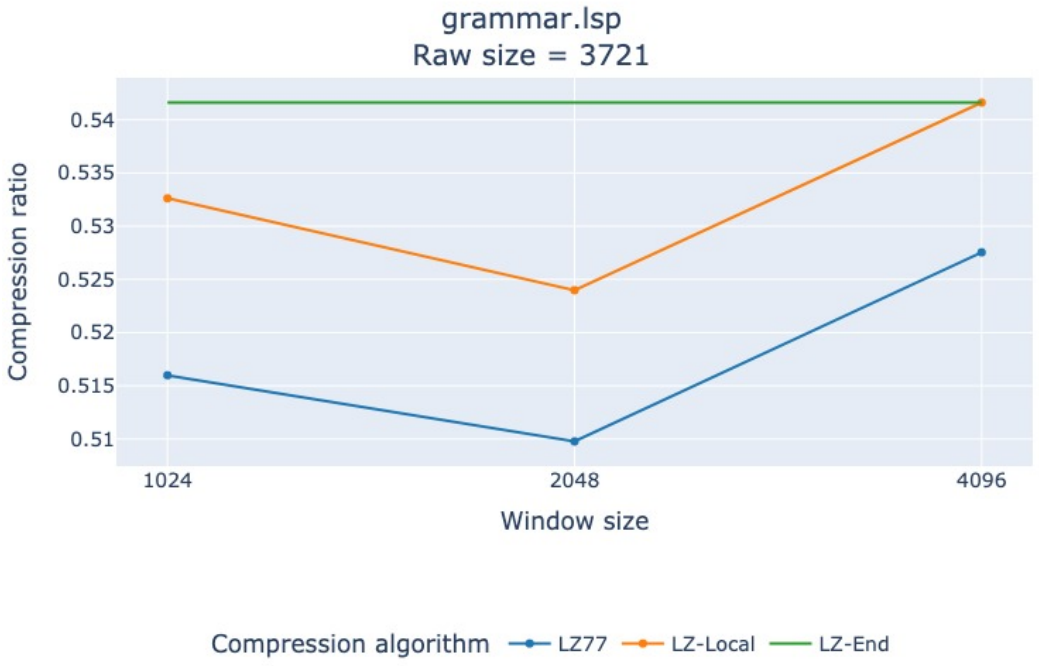


Figure 7.9: LZ-Local and LZ-77 have similarly-shaped graphs again; however, it is difficult to draw meaningful conclusions from such a small file.

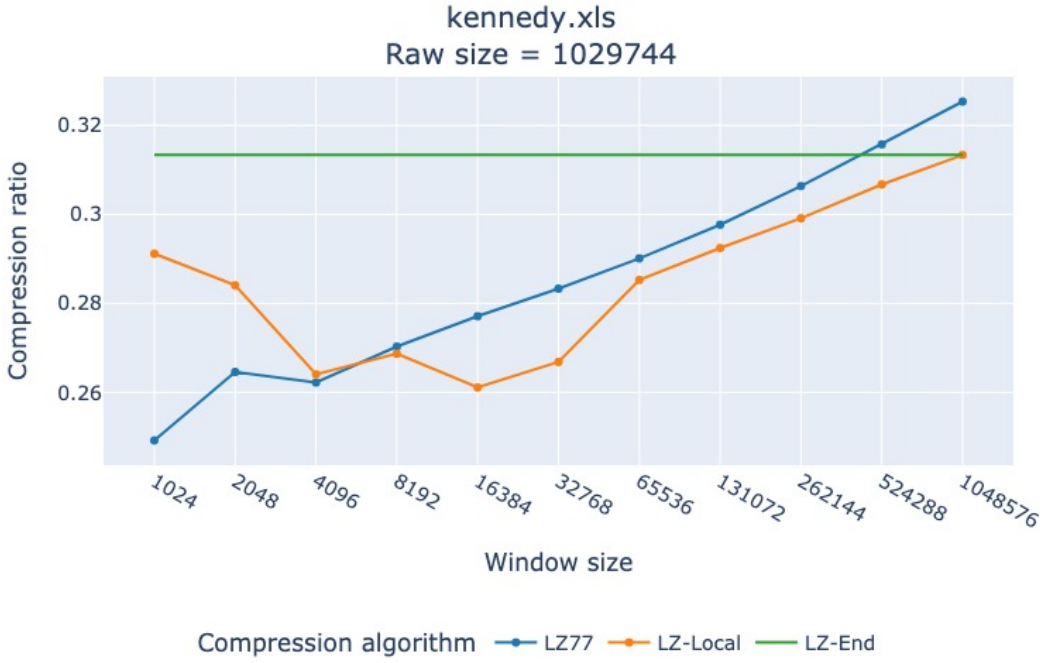


Figure 7.10: Here is the only example where LZ-Local outperforms LZ-77 for larger window sizes. Notice, however, that LZ-77 still produces the best compression ratio, when used with the smallest window size of 1024 symbols.

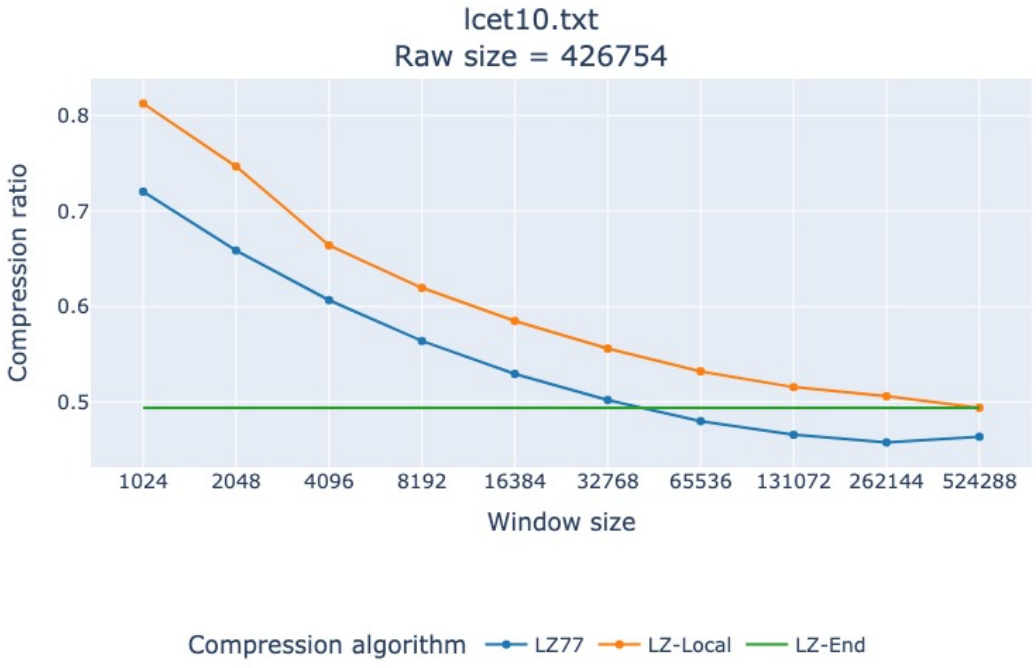


Figure 7.11: This file continues the trend observed for medium-entropy, non-repetitive files.

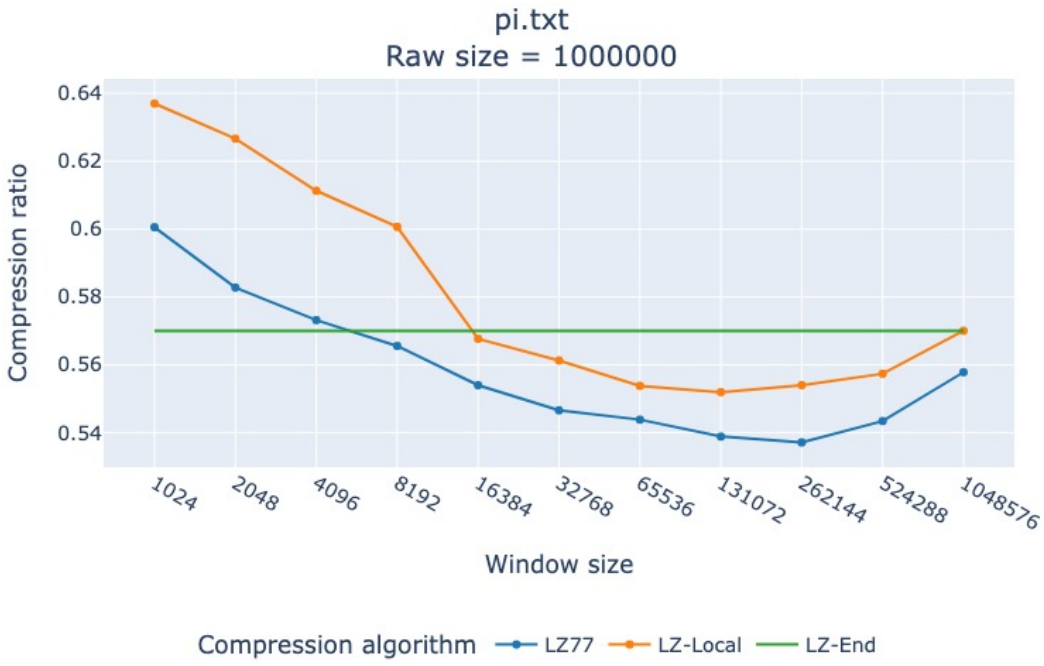


Figure 7.12: The LZ-Local and LZ-77 compression ratio have a similar relationship to window size for this file. Note that the larger window sizes harm the compression ratio – this is because the patterns in the file are more localised, meaning that a larger window size does not provide additional benefits to the compression.

Figure 7.13: This file continues the trend observed for medium-entropy, non-repetitive files such as `lcet10.txt` and `pi.txt`.

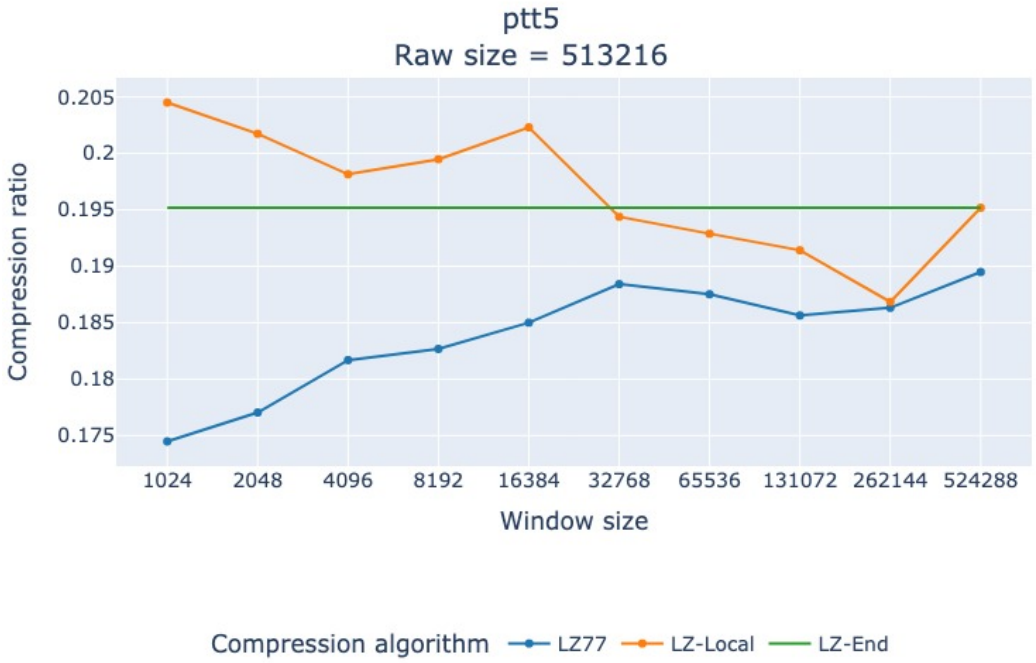


Figure 7.14: This file represents an exception, in that, where different window sizes have different effects on the LZ-Local compression ratio compared to LZ-77. Why this is so will require future investigation.

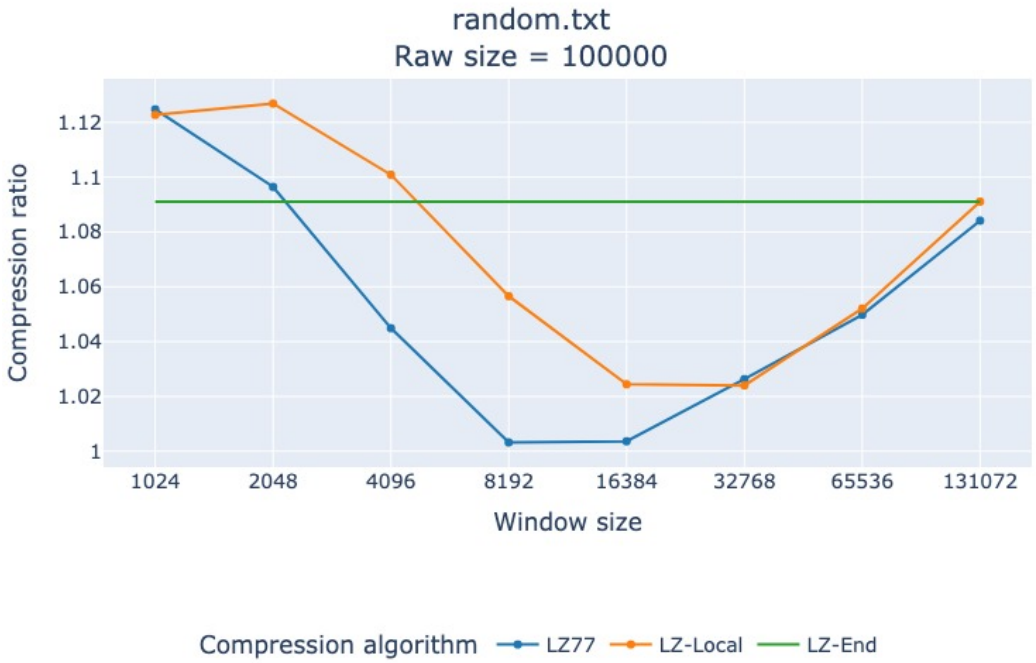


Figure 7.15: A random file will not compress; however, the graph shows that a larger window size does limit the number of LZ-Local phrases, reducing its compression ratio. This continues until a window size of 32768, after which larger window sizes increase the phrase size without effectively reducing the number of phrases into which the string is compressed.

Figure 7.16: This file resumes the trend observed for medium-entropy, non-repetitive files (such as `bible.txt` and `lcet10.txt`), where LZ-Local and LZ-77 exhibit a similar relationship between compression ratio and window size.

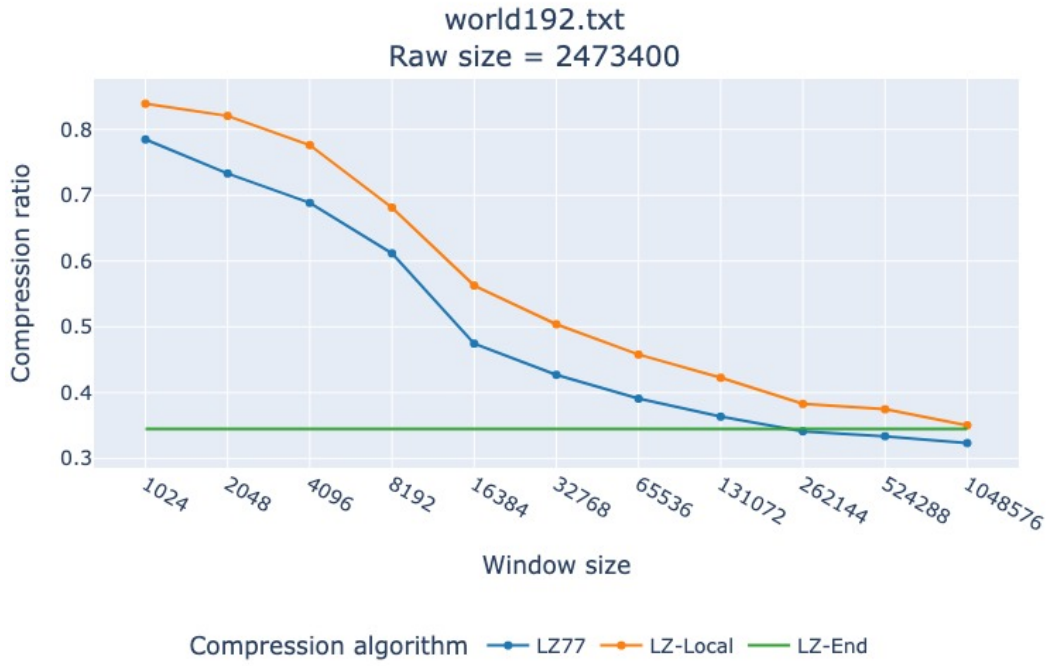


Figure 7.17: Again, LZ-Local and LZ-77 exhibit a similar relationship between compression ratio and window size.

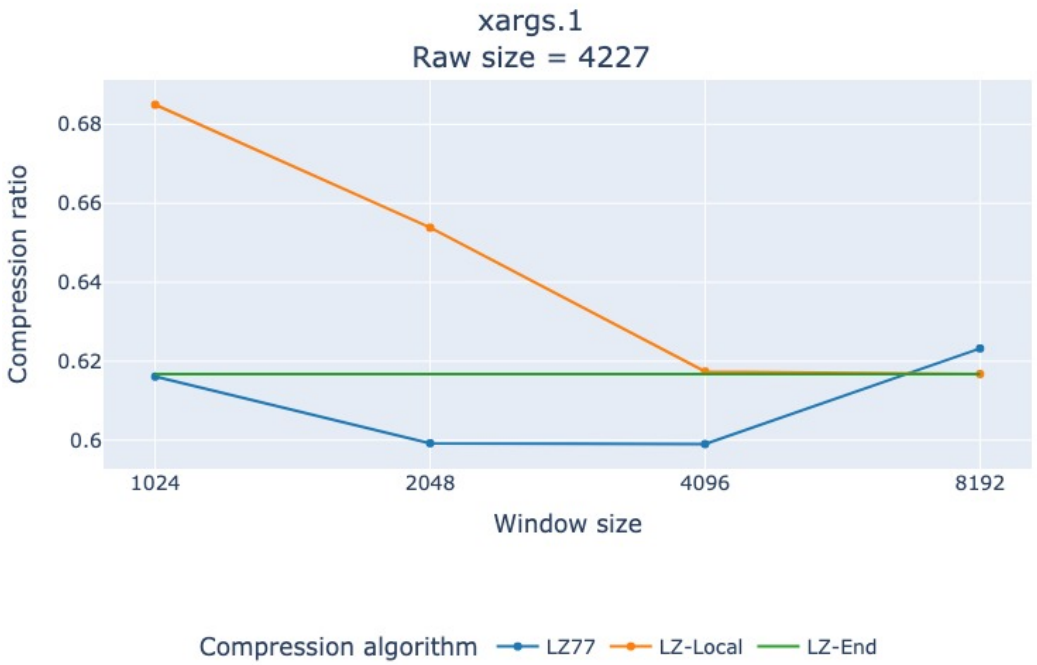


Figure 7.18: Here is another example where LZ-77 has poor performance for a larger window size. However, this is due to the phrase size of LZ-77 increasing for the larger window size, but without having a sufficiently long string to compensate for this loss.

7.2.2 Calibrated entropy strings

Recall Figure 3.1 (repeated for reference), which shows the LZ-77 compression ratio for each of the 10 entropy calibrations. The equivalent graph for LZ-Local is given in Figure 7.19. Following the method described in Section 5.2, there are 30 strings of each entropy calibration, and each string is 5MB (5×2^{20} bytes) in length. Figures 3.1 and 7.19 each show the mean compression ratio of the 30 strings for each entropy calibration.

The LZ-77 mean curves generally have smooth slopes that consistently flatten as the window size increases. The LZ-Local curves, however, sometimes have inconsistent "jumps" in compression ratios for successive window sizes. Consider, for example, the entropy calibration 0.025, which has a "jump" in compression ratio between window sizes 65536 and 131072 (see the top curve in Figure 7.19).

The reason the LZ-77 curves are smooth, while the slopes of the LZ-Local curves are not, is due to the two parsings' different interpretation of back-references: Recall that LZ-77 back-references are *symbol counters* (Section 3.2). This means that the number of bits required to store each LZ-77 back-reference is exactly the number of bits required to store the size of the sliding window.

In contrast, the LZ-Local back-references are *phrase counters* (Section 7.1).

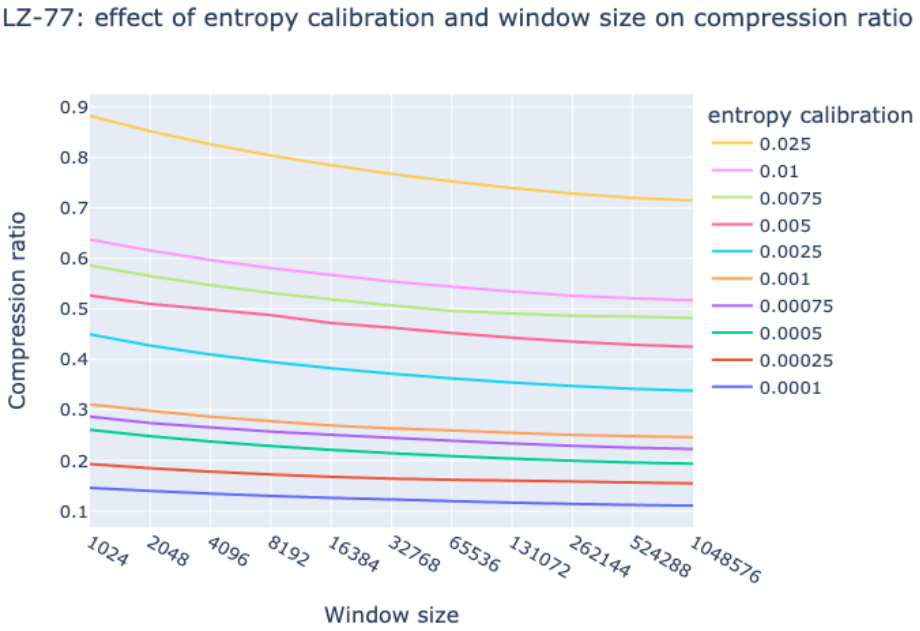


Figure 3.1: The mean compression ratio achieved by LZ-77 on the strings with 10 different entropy calibrations, for various window sizes. Note the consistent relationship between larger window size and improved compression, for all entropy calibrations.

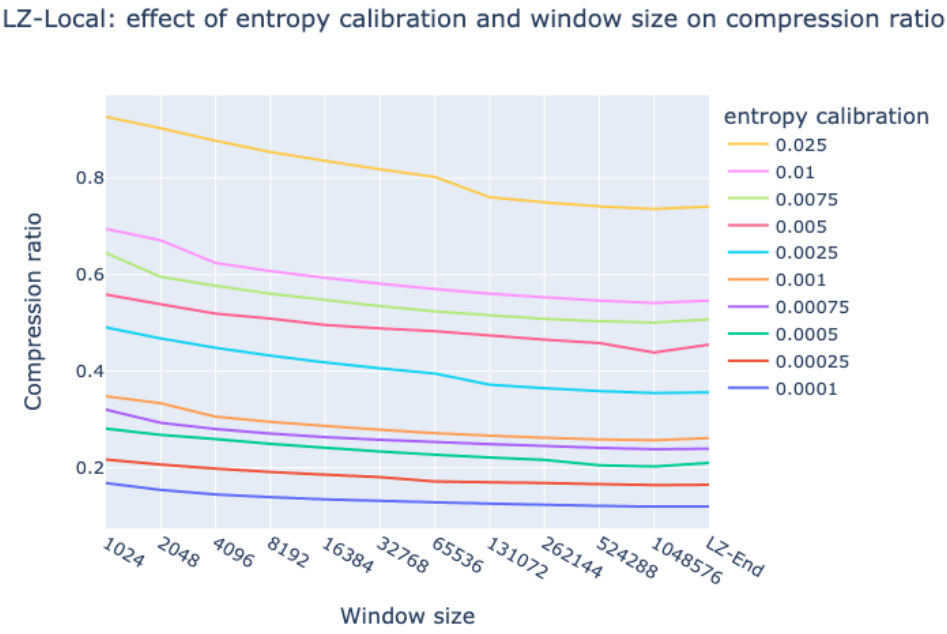


Figure 7.19: The mean compression ratio achieved by LZ-Local on the strings with 10 different entropy calibrations, for various window sizes. We include the LZ-End compression ratio on the right-hand side of the graph. Note that there is minimal difference in compression between LZ-End and LZ-Local, when the window size is $\geq 2^{17} \approx 130{,}000$. Also note that the slopes of the LZ-Local curves are not as smooth or consistent as for LZ-77 in the figure above. This is because the LZ-Local phrase sizes are only indirectly tied to the window size, while LZ-77 phrase sizes are directly tied to the window size.

This means that the number of bits required to store the back-reference is only indirectly affected by the window size: the number of phrases required to encode w symbols is $\leq w$. Therefore, incrementing the number of bits required to store the window size *will not necessarily* increment the number of bits required to store the phrase back-reference.

If we increment the number of bits storing the window size, and this increments the size of LZ-Local back-references, there will be relatively little change in compression ratio between the two window sizes. However, if incrementing the size of the window does not result in an increase in the size of the LZ-Local back-reference, there will be a larger gain in compression ratio.

Figures 7.20 to 7.29 compare the three parsing algorithms for each entropy calibration. These graphs show the mean compression ratio for each algorithm, with bands showing the minimum and maximum compression ratios achieved for each window size. The LZ-End "curves" are actually each single points, since LZ-End does not have a sliding window.

In general, the graphs show that the LZ-77 compression comfortably outperforms LZ-Local for all window sizes, and outperforms LZ-End for larger window sizes. This matches our observations from the Canterbury files in the previous section. The LZ-Local compression rate approaches LZ-End as the window size increases, and for the largest window size (2^{20} bytes) often outperforms LZ-End. This is interesting, as the largest window size is one fifth of the raw size (5×2^{20} bytes) of each calibrated entropy string. This evidence supports our hope that LZ-Local compression could in fact be coarsely optimal.

Another important observation from Figures 7.20 to 7.29 relates to the variance in observed compression ratios. Consider Figure 7.27: For some window sizes, the minimum and maximum compression ratios for LZ-77 and LZ-Local are so close together, that the "bands" around the mean value basically disappear. For some window sizes, on the other hand, the bands return, and the mean value is close to the bottom of the band.

The reason for this is that the compression ratio is influenced by two variables: 1) the number of phrases which the compression algorithm parses the input string into, and 2) the size of each phrase. In the case of all three compression algorithms, the phrases size has a much larger variance than the phrase count. This means that the variance in compression ratio is therefore impacted by the variance in phrase size much more so than by the variance in phrase count. Section 7.2.2.1 explains this in detail.

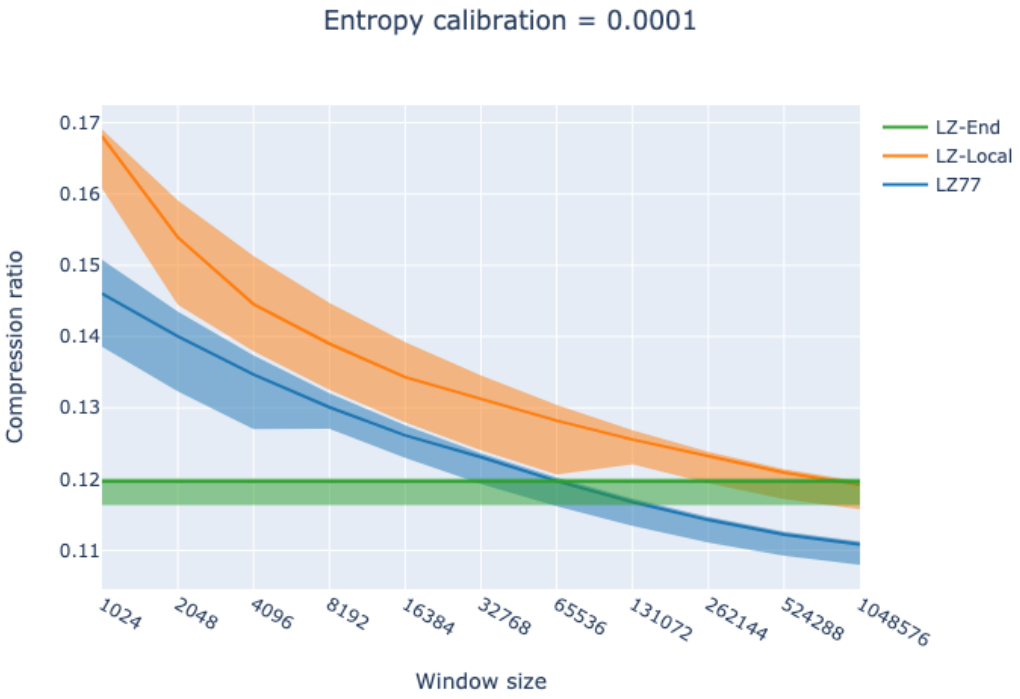


Figure 7.20: Comparing the LZ-Local compression ratio to that of LZ-End and LZ-77. This is the lowest entropy calibration. The graph shows the min-mean-max for the 30 files for each window size. Note that LZ-77 is superior to LZ-Local for all window sizes, and that LZ-Local approaches LZ-End for larger window sizes.

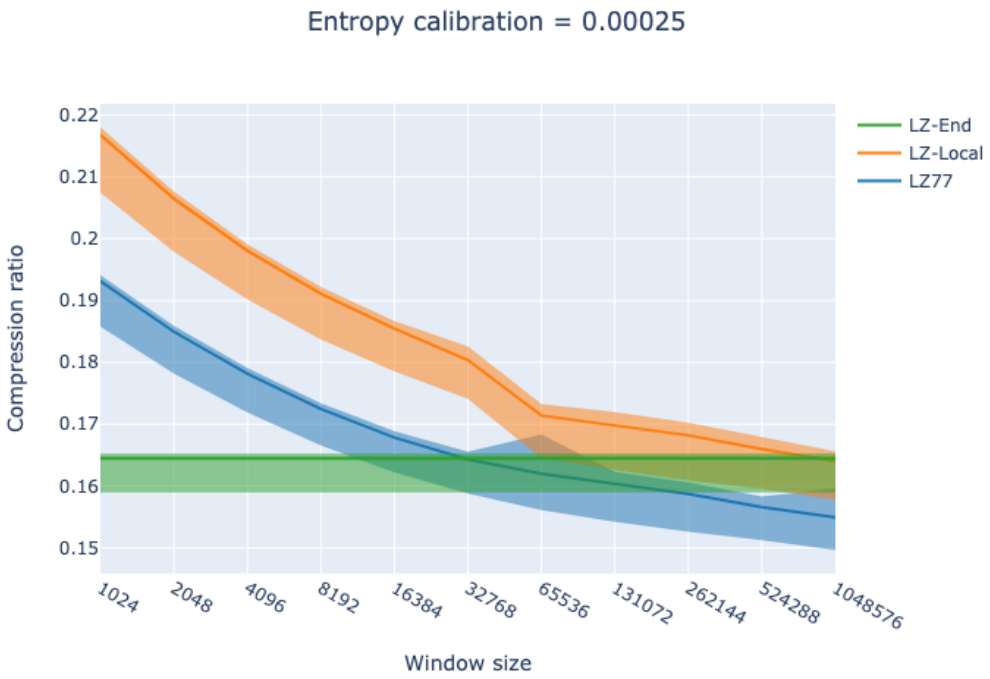


Figure 7.21: Comparing the LZ-Local compression ratio to that of LZ-End and LZ-77. This time there is some overlap between the maximum LZ-77 compression ratio and the minimum LZ-Local, for some window sizes. Note that for the largest window size, LZ-Local outperforms LZ-End. Note also the jump in LZ-Local compression ratios between window sizes 32768 and 65536, and see that this same jump is not present in LZ-77. This is a result of the direct correlation between window size and phrase size for LZ-77, compared to the indirect correlation for LZ-Local.

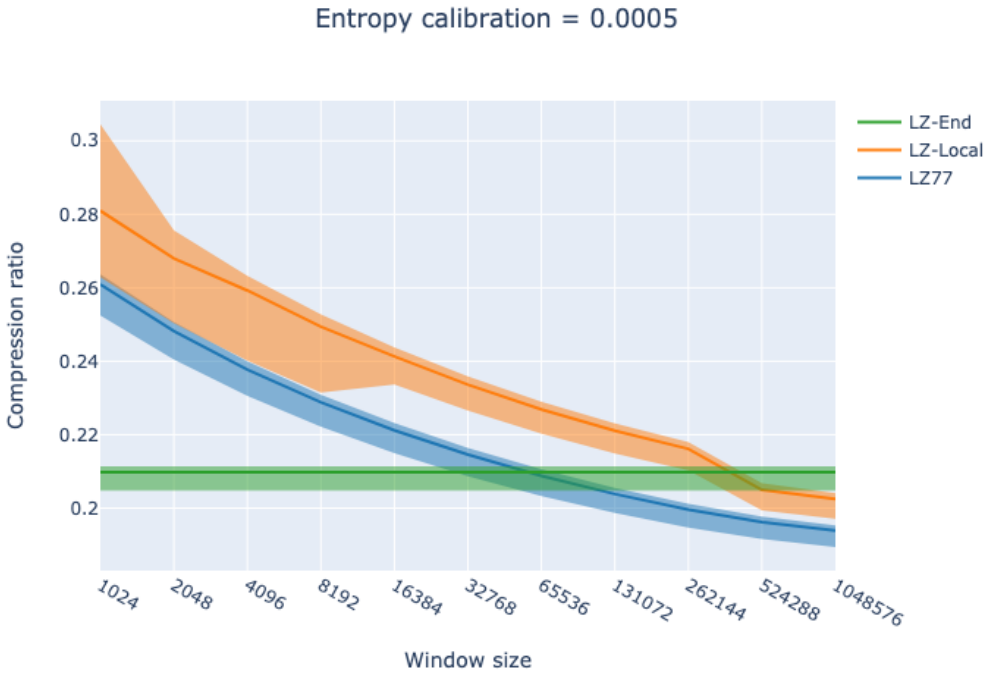


Figure 7.22: Comparing the LZ-Local compression ratio to that of LZ-End and LZ-77. LZ-Local outperforms LZ-End for the two largest window sizes.

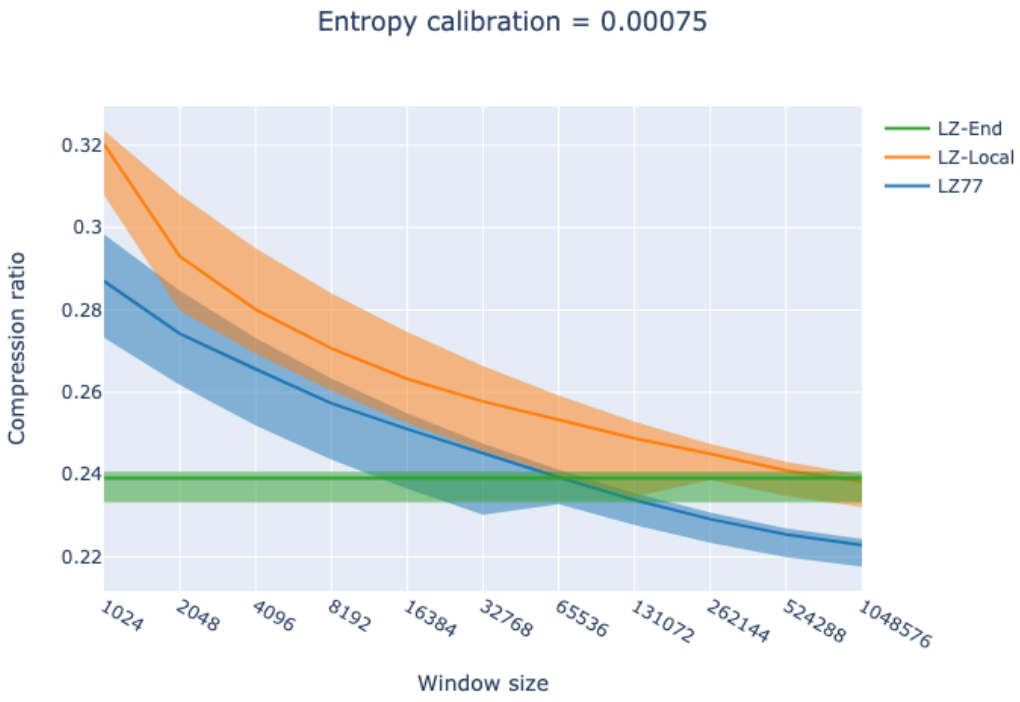


Figure 7.23: For this entropy calibration, the trend continues: LZ-77 outperforms LZ-Local for all window sizes, and LZ-Local converges on LZ-End for larger window sizes.

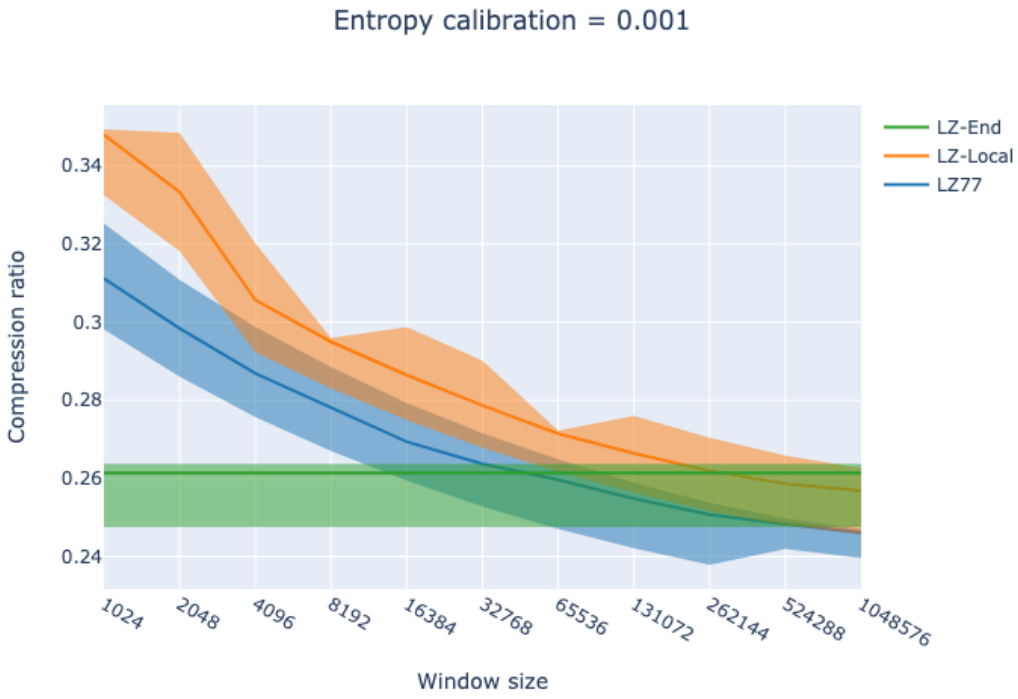


Figure 7.24: Again, the pattern repeats, where LZ-77 outperforms LZ-Local and LZ-Local outperforms on LZ-End, for large enough window sizes.

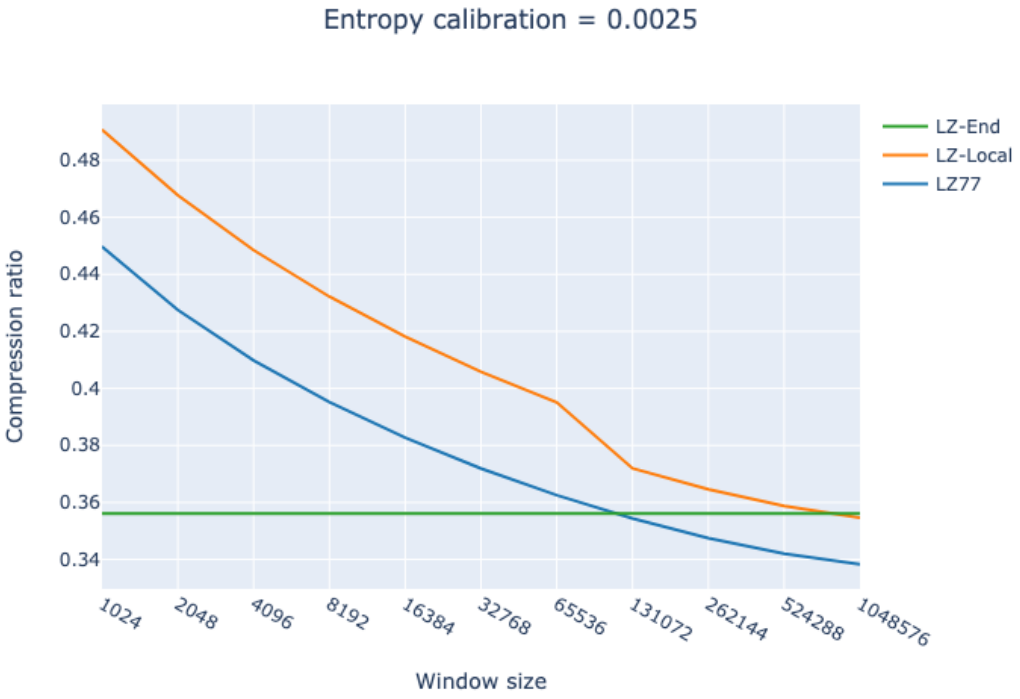


Figure 7.25: This entropy calibration continues the trend observed in previous figures, where LZ-77 outperforms LZ-Local, and the LZ-Local compression rate (marginally) outperforms that of LZ-End for the largest window size. Note the extremely consistent results here, where the minimum and maximum compression ratios for the 30 files are always very close to one another, and the variance bands observed for the other entropy calibration levels disappear here. This is because there is no variation in the phrase sizes for each window size. For the majority of entropy calibrations, the LZ-77 compression ratio crossed that of LZ-End, when the window size was between 32768 and 65536. Here, however, LZ-77 is slower to outperform LZ-End, requiring a window size $>$ 131072 to do so.

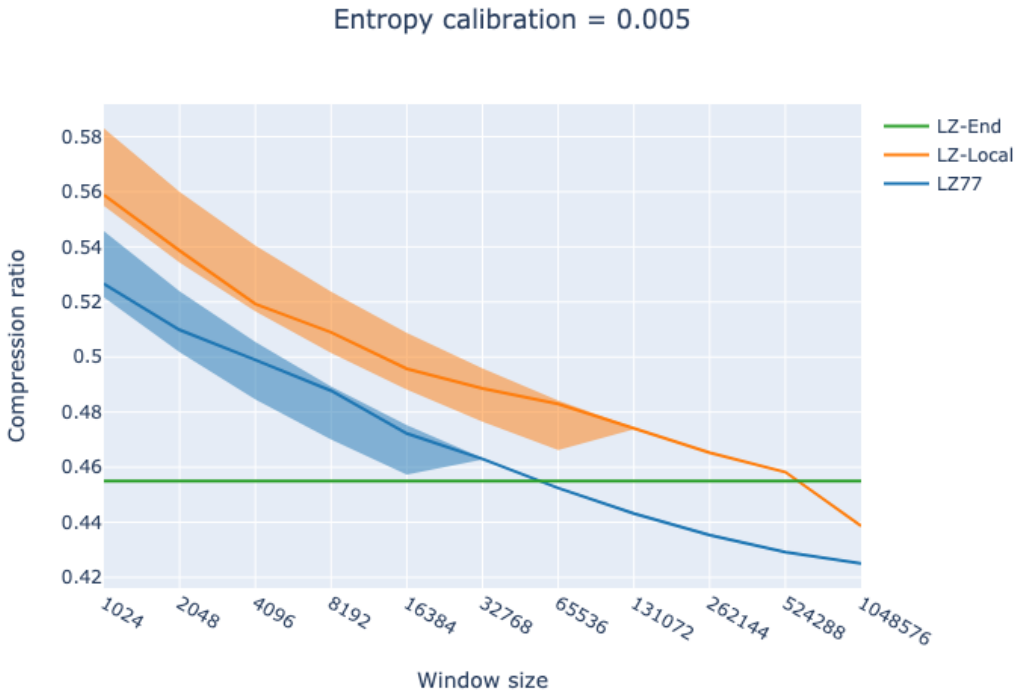


Figure 7.26: Again, LZ-77 outperforms LZ-Local for all window sizes. However, for the largest window size, LZ-Local outperforms LZ-End. For larger window sizes, the variance for LZ-Local and LZ-77 reduces. This again corresponds to a reduced variance in phrase size (Section 7.2.2.1). Note that the variance of the LZ-77 compression ratio reduces at a much smaller window size (32768) than for LZ-Local (131072).

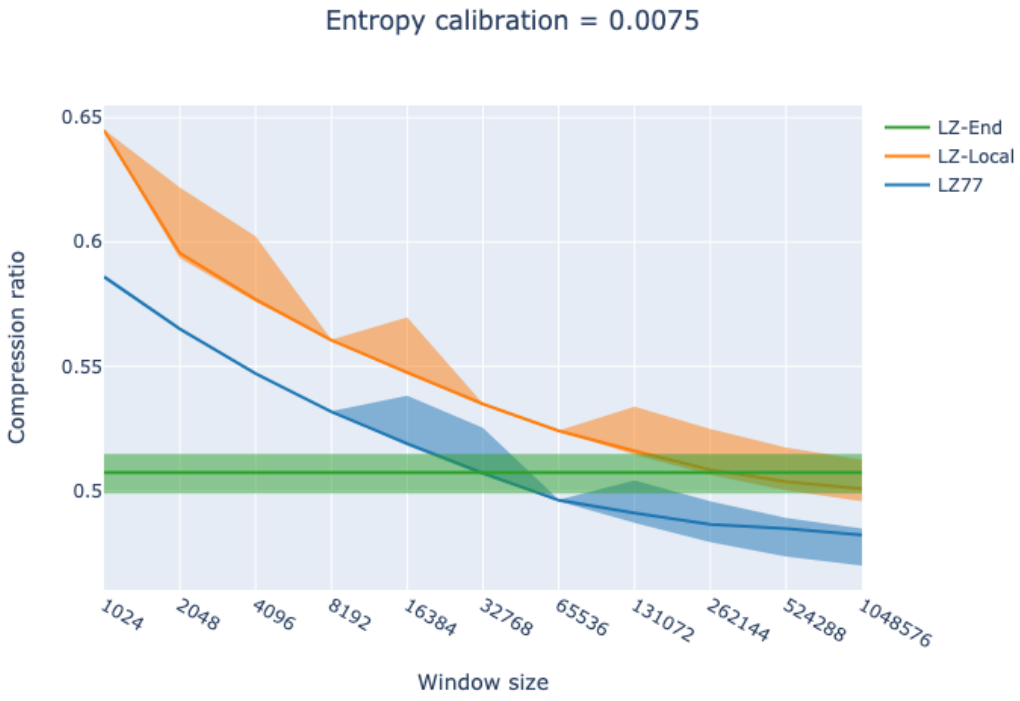


Figure 7.27: This graph shows similar relative performance between LZ-Local and LZ-77 as previous entropy calibrations. Note, however, that LZ-Local outperforms LZ-End for the two largest window sizes.

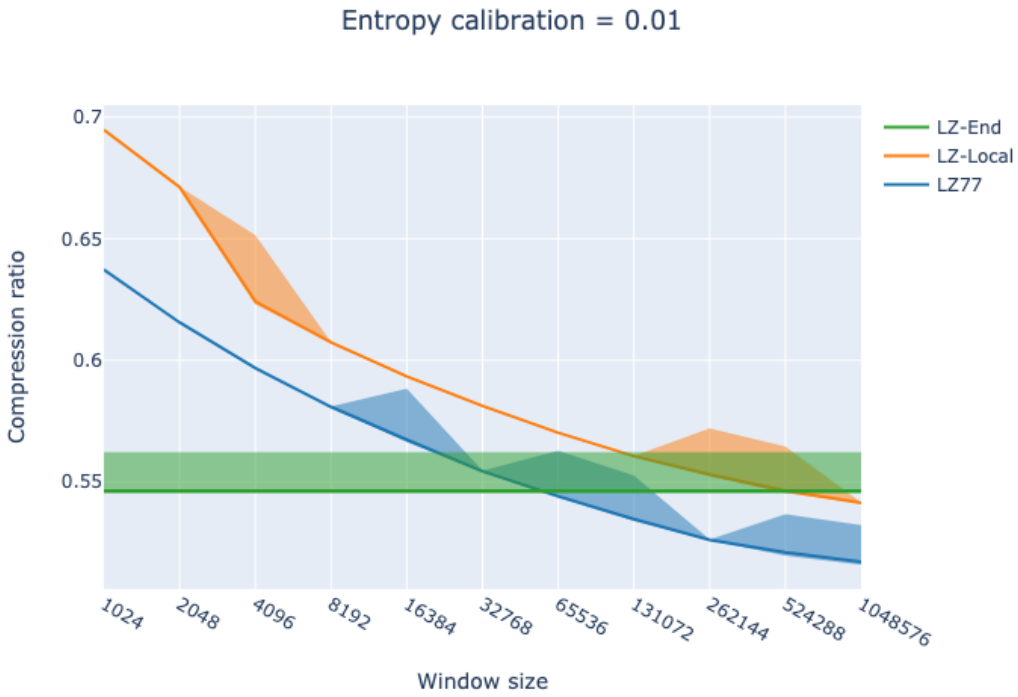


Figure 7.28: Once again, LZ-77 outperforms LZ-Local, and LZ-End for large enough window sizes. Note that the outlying compression ratios for LZ-Local and LZ-77 do not correlate. For example, LZ-77 has an outlying poor compression ratio for one file at window size 16384, while the LZ-Local compression values for this window size are very consistent. Conversely, LZ-Local's outlying value at window size 4096 is not replicated in LZ-77. This is because the outliers are caused by outliers in the size of each phrase; we do not expect minor differences in LZ-77 phrase size to correspond to differences in LZ-Local phrase size (or vice versa).

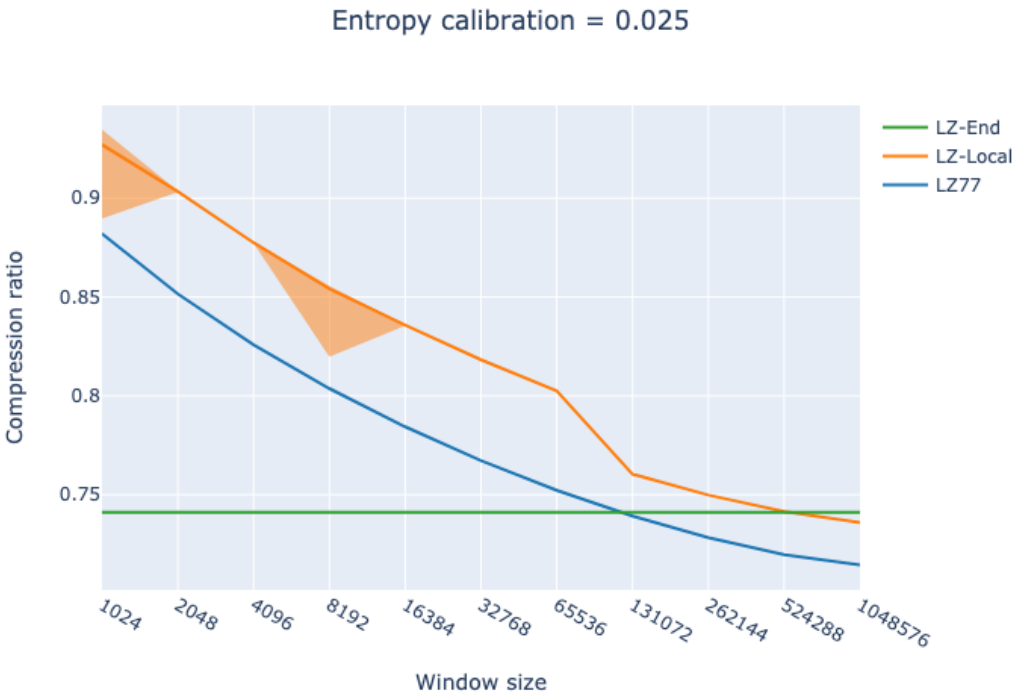


Figure 7.29: For the highest entropy calibration, the variance is in compression ratios is very small, save for a couple of outliers in LZ-Local. The trend repeats, with LZ-77 outperforming LZ-Local for all window sizes. Note that for this entropy calibration along with $\epsilon = 0.0025$ (Figure 7.25), LZ-77 only outperforms LZ-End for window sizes > 131072. This is different to the other entropy calibrations, where LZ-77 outperforms LZ-End for window sizes greater than 32768 or 65536.

7.2.2.1 Cause of the variance in compression ratios

This section demonstrates that the distribution of compression ratios for LZ compressors is more reliant on the distribution of phrase sizes than on the distribution of phrase counts. To do this, we focus on the entropy calibration 0.005, compressed by LZ-77 using a window size of 2048 (Figure 7.26, repeated below).

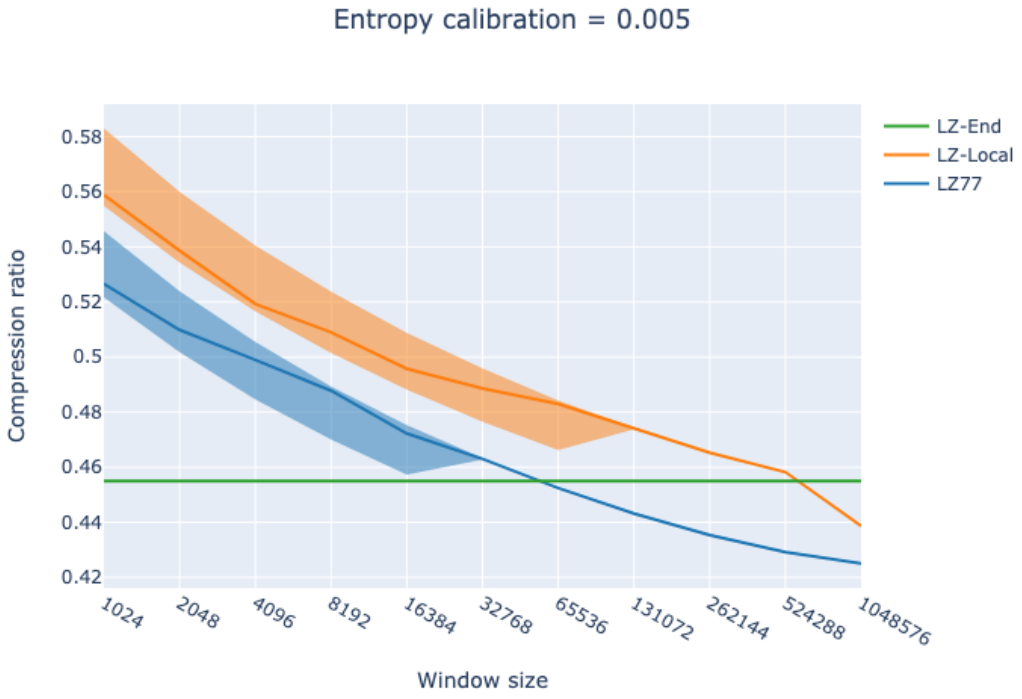


Figure 7.26: Repeating the figure for reference. This section investigates the distribution of LZ-77 compression values.

We are interested in the distribution of compression ratios within the min-max bands. To this end, we plot the LZ-77 compression ratio compared to window size as a box-and-whisker plot for each window size (Figure 7.30). In a similar vein, Figures 7.31 and 7.32 show the phrase count and compressed phrase sizes, as box-and-whisker plots. The phrase count has very little variation for any window size. There are, however, variations in the size of each phrase, and this corresponds to the window sizes with most variation in compression ratio.

To prove beyond all doubt the relationship between the variance in phrase size, phrase count, and compression ratio, we generated 1000 additional strings with the same entropy calibration (0.005) and size (5×2^{20} bytes), and compressed these using LZ-77 with a window size of 2048. The phrase counts of these 1000 strings have a largely unimodal distribution (Figure 7.33). There are two phrases sizes observed (Figure 7.34). The interesting graph is the distribu-

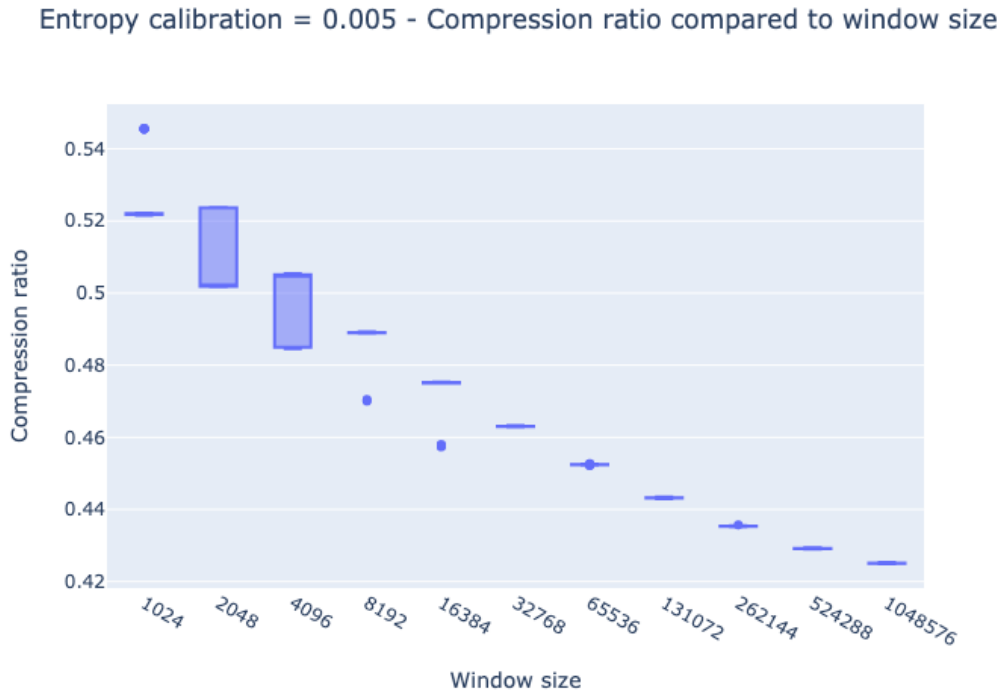


Figure 7.30: Box-and-whisker plots of LZ-77 compression ratio. This isolates the LZ-77 compression ratio from Figure 7.26.

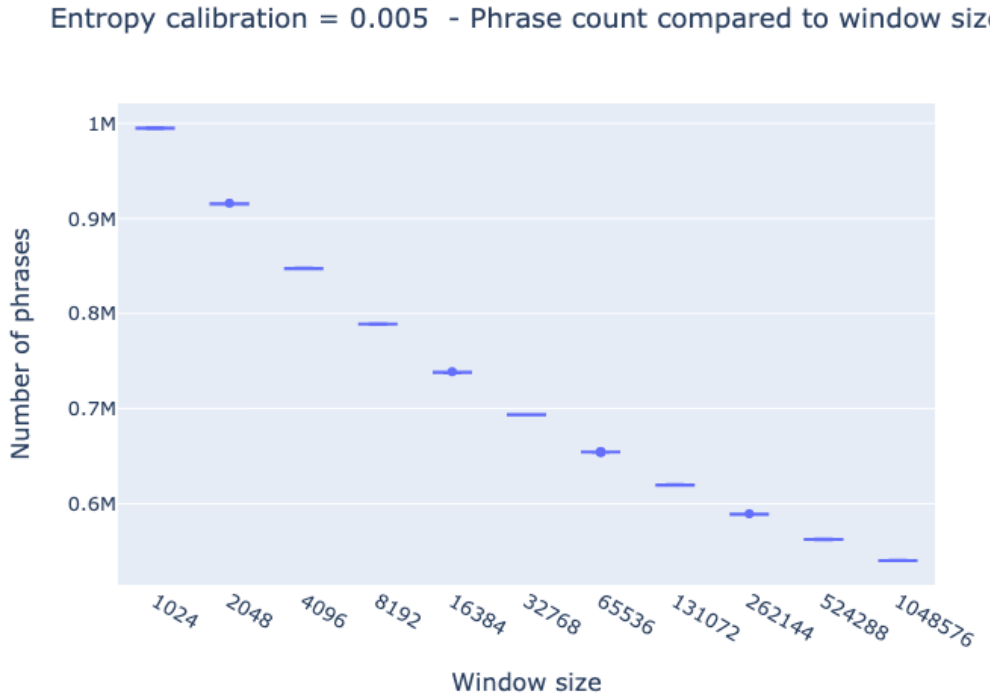


Figure 7.31: Box-and-whisker plot showing the distribution of phrase counts compared to window size. The number of phrases has very little variance compared to the compression ratio from Figure 7.30.

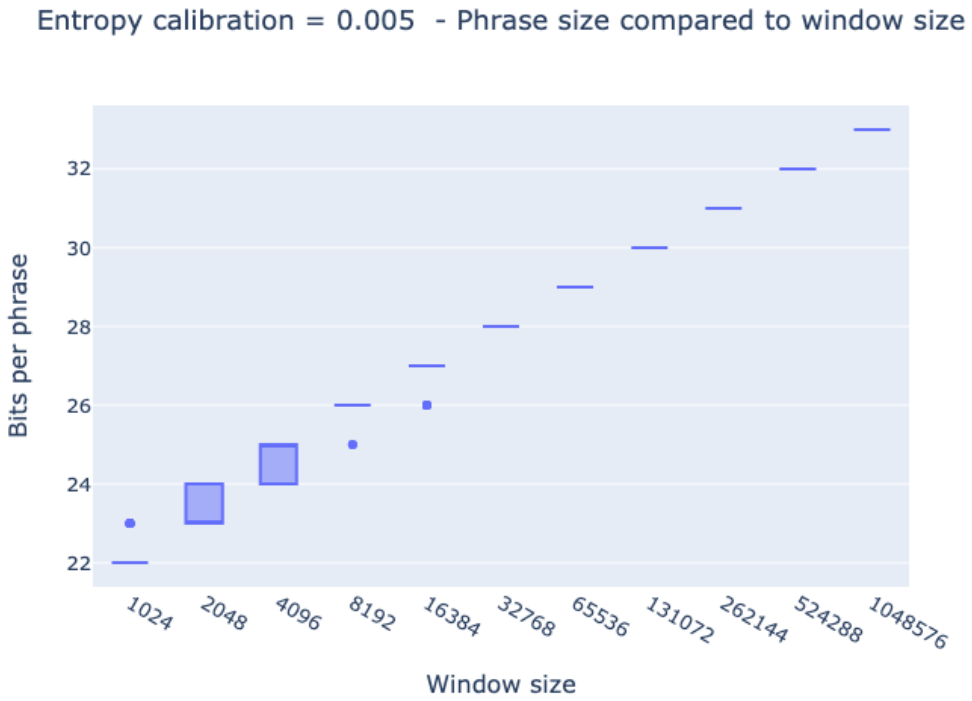


Figure 7.32: Box-and-whisker plot showing the number of bits encoding each LZ-77 phrase. The window sizes 32768 and larger have no variation in phrase size, and this corresponds to the reduced variance in compression ratio for larger window sizes (Figure 7.30). The outlying phrase sizes (signified by the dots at window sizes 1024, 8192 and 16384) correspond to outlying compression ratios in Figure 7.35.

tion of compression ratio (Figure 7.35), where the distribution of compression ratio is bi-modal, corresponding to the two phrase sizes! This proves that the variance in compression ratio is predominantly influenced by variance in the size of each phrase, rather than by variance in the phrase counts.

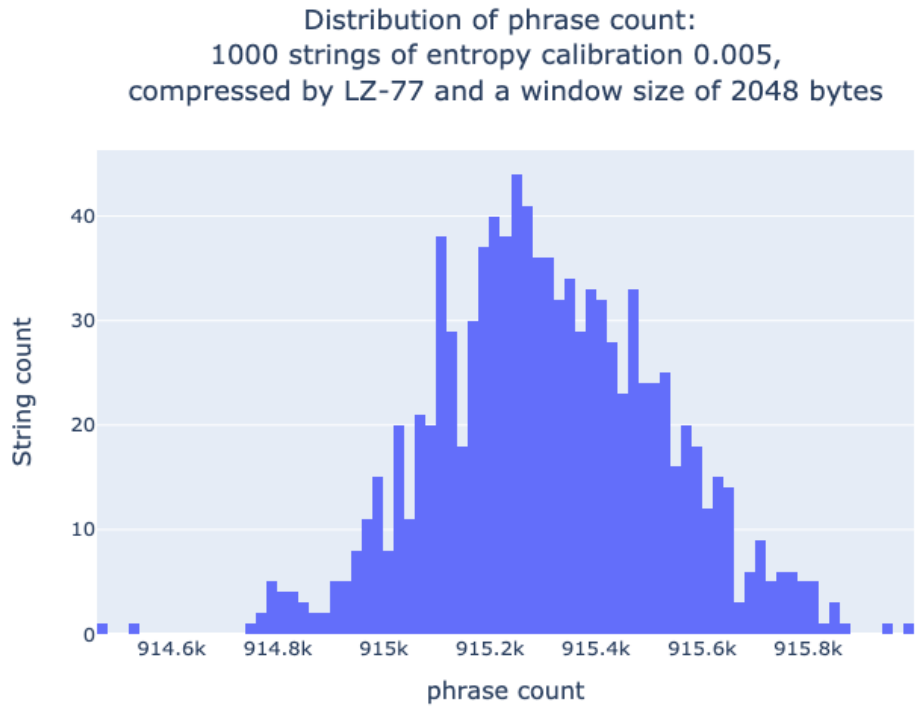


Figure 7.33: Distribution of LZ-77 phrase counts, when compressing 1000 strings of calibrated entropy. There is nothing particularly interesting about this graph, except that the distribution only has one peak.

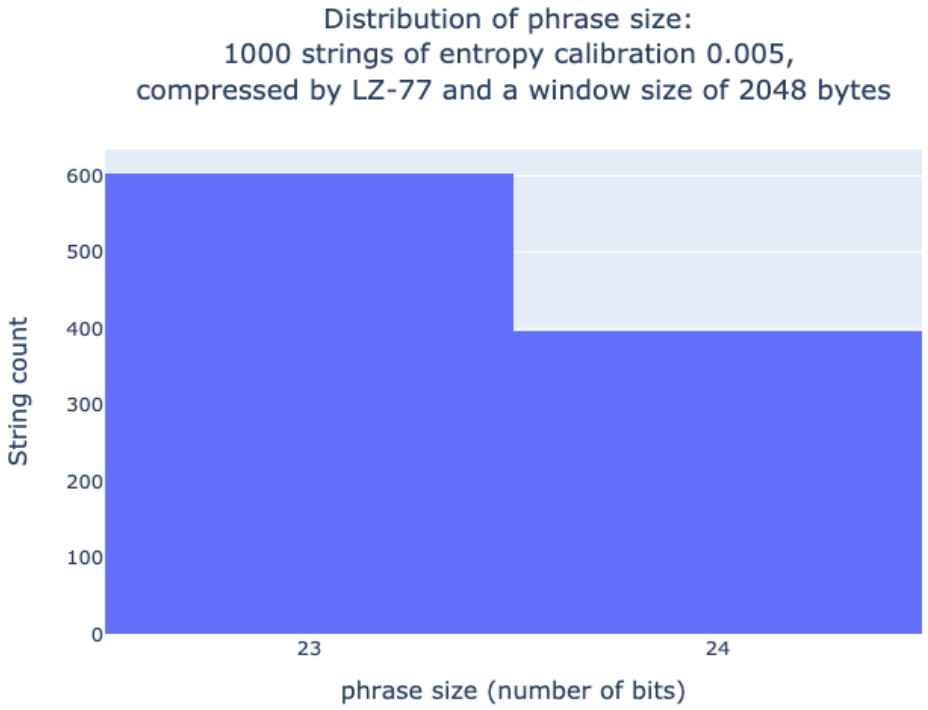


Figure 7.34: There are only two different phrase sizes observed across these 1000 strings, with approximately 60% of phrases stored in 23 bits, and 40% stored in 24 bits.

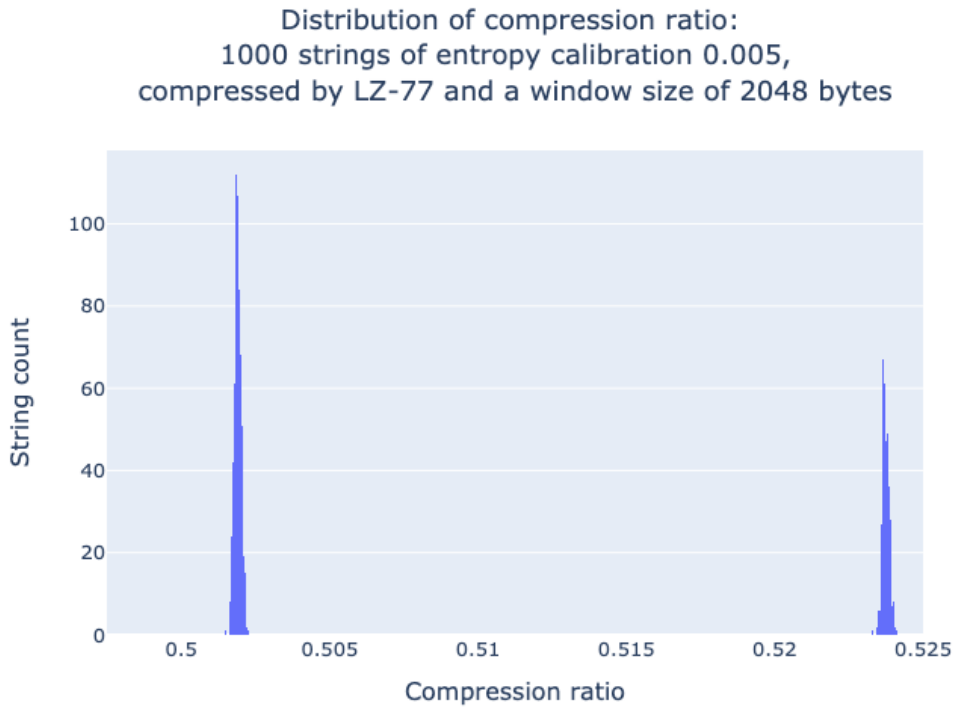


Figure 7.35: The distribution of compression ratio for the 1000 strings. Even though the phrase count had only one mode to its distribution, the compression ratio has two modes! These correspond to the two phrase sizes observed in Figure 7.34.

7.3 Editing algorithm for LZ-Local

This section adapts the random editing algorithm for LZ-End (Algorithm 8) for use on the LZ-Local parsing. While we use the LZ-End editing algorithm as the base for editing LZ-Local parsing, we need to make a number of changes: 1) We can optimise the runtime of the editing algorithm to make use of the sliding window, and 2) We need to make sure that the edit does not stretch any phrase's back-reference beyond the window limit.

The general outline of the editing algorithm for LZ-Local is as follows, where w is the window size used to construct the LZ-Local compression:

- Identify the phrases to edit.
- Identify dependent phrases. This search can now be restricted to those symbols within w symbols of the final edited symbol.
- Identify phrases whose back-references would be extended beyond the window limit w. This step is only necessary if the edit operation inserts more symbols than it removes.
- Decode the target phrases and the w symbols which precede and follow them.
- Edit the decoded string.

- Using the phrase boundaries in the w symbols preceding the first target phrase, reconstruct the internal state of the LZ-Local parser, and use this to parse the edited part of the decoded string.
- For each phrase which follows the target and appears within the decoded string:
  - If the phrase is a dependent or extended back-reference, use the state of the LZ-Local parser to parse the decoded symbols.
  - Else, add the phrase to the state of the LZ-Local parser, and adjust the phrase's back-reference to account for the number of phrases we have replaced.
- For the next w symbols following the decoded string: adjust the back-references of each phrase, if necessary.

Remark 7.4. Note that `LZ-End edit()` (Algorithm 8) parses the inserted string independently of the existing parsing. The editing algorithm for LZ-Local, however, uses the existing parsing to parse the inserted string.

The sliding window in the LZ-Local parsing enables this by limiting the cost (both time and memory) of reconstructing the state of the LZ-Local parser. This state can then be used to parse the inserted string. Doing the same for LZ-End would be equivalent to decoding and re-parsing the entire string to the left of the edit location; doing this would not be considered random editing, as explained in Remark 6.6.

Sections 7.3.1 to 7.3.4 explain individual components of the editing algorithm, and Section 7.3.5 then ties these components together into the complete editing algorithm.

7.3.1 Incremental parsing

The focus of this thesis is not on how to construct the LZ-Local parsing. We do, however, have to define the signature of the `LZ-Local incremental parse()` function (Algorithm 9): Instead of parsing the entire string, this function parses only a substring thereof. This enables the editing algorithm to parse the inserted string using the phrases which occur to the left of the target phrases.

The function takes the following arguments:

- String S: The string which we wish to partially parse.
- LZ-Local parsing Π: The LZ-Local phrases representing the prefix of S which has been partially parsed.
- Integer w: The size of the sliding window used by the LZ-Local parser.
- Integer f: The first symbol in S which must be parsed. This means that Π contains the parsing of $s_{0,f-1}$ (or $\emptyset$, if $f = 0$).

- Integer l: The number of symbols in S to parse.

Note that the function definition (Algorithm 9) states that the return value is the parsing of $s_{f,f+l-1}$, **not** the entire parsing. The reason for doing this will become clear in Section 7.3.5, which uses this function as part of the LZ-Local editing algorithm.

The first step of the algorithm will reconstruct the internal state of the LZ-Local parser. How this is done will depend on the implementation of the LZ-Local parsing algorithm. However, with the raw string S and the phrase boundaries in Π, it is possible to reconstruct this state. The sliding window of LZ-Local means that this state can be reconstructed *in no more time than is required to parse a string of length* w.

This state must be constructed for the first call to the `incremental parse()` function, but can be stored for future calls to the function.

Algorithm 9 `LZ-Local incremental parse`(S, Π, w, f, l)

Require: String S: A string which we wish to partially parse.
LZ-Local parsing Π: The parsing of the prefix of S.
Integer w: The size of the sliding window.
Integer f: The first symbol in S which has not been parsed into Π.
Integer l: The number of symbols in S to parse.
Result: Array Φ: The LZ-Local phrases representing the symbols $s_{f,f+l-1}$.

Comment: only the symbols $s_{f-w,f-1}$ *(and the suffix of* Π *which encodes these symbols) are required to construct the state of the LZ-Local parser. This state can be stored and re-used for future calls to the function.*

Remark 7.5. The authors have implemented the `incremental parse()` function for both the LZ-Local and LZ-End parsing functions. In the case of LZ-End (Algorithm 2), constructing the internal state of the parser involves calculating the suffix array, inverse suffix array, and BWT of the reverse of S, calculating the array C over the BWT, and using the phrase boundaries of Π to build the structure Λ (which maps the innovation symbol components of parsed phrases to entries in the inverse suffix array).

7.3.2 Identify dependent phrases

Identifying the dependent phrases of LZ-Local is functionally similar to that of LZ-End. We provide the details of this in Algorithm 10. The two algorithms (Algorithm 4 for LZ-End and Algorithm 10 for LZ-Local) take the same parameters, with the exception that the LZ-Local variant has one additional parameter: the size of the sliding window.

Algorithm 10 `LZ-Local identify dependent phrases`(Π, w, a, b, suf_len)

Require: LZ-Local parsing Π
Integer w, denoting the sliding window size used to construct Π.
Integers a, b, denoting the first and last target phrases.
Integer suf_len, denoting the number of symbols at the end of π_b not considered part of the target phrase.

Result: Array D of pointers to phrases in Π which back-reference one or more phrases in $\pi_{a,b}$.

```
1:  D ← ∅
2:  ptr ← b
3:  limit ← minimum(Π.size, rank(select(b) + w))
4:  while ptr < limit do
5:      ref ← ptr − π_ptr.b
6:      if a ≤ ref < b and π_ptr.l > 0 then
7:          D ← D || ptr
8:      else if ref ≥ b then
9:          len ← π_ptr.l
10:         while len > 0 do
11:             if ref = b and len > suf_len then
12:                 D ← D || ptr
13:                 break
14:             end if
15:             len ← len − (π_ref.l + 1)
16:             ref ← ref − 1
17:         end while
18:     end if
19:     ptr ← ptr + 1
20: end while
21: return D
```

Note that the sliding window in LZ-Local permits the early exit condition at Lines 3 and 4.

The algorithm presented here is rather naïve: There is no storing of the distance of a phrase from the target, like we do for the equivalent algorithm in LZ-End (see Algorithm 4). Incorporating this optimisation is trivial, given the reference implementation in Algorithm 4. Such an optimisation will reduce the `while` loop at Line 10 to a single step.

Assuming this optimisation has been made, the algorithm executes in time $\Theta(w)$, plus the cost of a `rank` and a `select` operation. The memory requirements are $O(w)$, since there are at most w dependent phrases, and storing the distances of phrases to the target requires an integer array with w elements.

7.3.3 Identify extended back-references

A major point of difference between the LZ-Local and LZ-End editing algorithms relates to the phrases following the target phrases which need to be replaced: For LZ-End, only the first-order dependent phrases need to be replaced. For LZ-Local, there is another class of phrases which need to be replaced: these are the phrases whose back-references are extended beyond the limits allowed by the sliding window. This can happen when the edit operation inserts more symbols than it deletes. In this case, there is the risk that the edit results in some phrases being further from their back-reference than the w symbols allowed by the sliding window.

Algorithm 11 identifies these phrases. Note that we apply this algorithm *prior to* applying the edit. That is, we use the size of the edit operation to determine the phrases in Π whose back-references *will* be extended beyond the window size, once we have applied the edit.

The algorithm first checks whether or not the edit inserts more symbols than it removes. The algorithm only continues if the edit will increase the size of the raw string (Line 2). Otherwise, the algorithm simply returns an empty array.

The variable *distance* stores the number of symbols whose phrases have been processed, while p is a pointer to the next phrase whose back-reference

Algorithm 11 `LZ-Local identify extended references`(Π, w, a, b, d)

Require: LZ-Local parsing Π, prior to applying the edit.
Integer w denoting the sliding window size used to construct Π.
Integers a, b, denoting the range of phrases in Π which have been edited.
Integer d, denoting the difference between the number of symbols which will be inserted and deleted by the edit operation.
Result: Array E denoting the indices in Π whose references have been extended beyond the window size w.

1: $E \leftarrow \emptyset$
2: **if** $d > 0$ **then**
3: $\quad$ *distance* $\leftarrow$ `select`$(b) - (\texttt{select}(a-1)+1)$
4: $\quad$ $p \leftarrow b$
5: $\quad$ **while** *distance* $< w$ **and** $p < \Pi.size$ **do**
6: $\quad\quad$ **if** $p - \pi_p.b < a$ **and** $(\texttt{select}(p) - \texttt{select}(\pi_p.b) + d) > w$ **then**
7: $\quad\quad\quad$ $E \leftarrow E \mid\mid p$
8: $\quad\quad$ **end if**
9: $\quad\quad$ $p \leftarrow p + 1$
10: $\quad\quad$ *distance* $\leftarrow$ *distance* $+ \pi_p.l + 1$
11: $\quad$ **end while**
12: **end if**
13: **return** E

must be checked for extension.

A phrase's back-reference can only be extended by an edit if it references a symbol prior to the target. Line 6 determines whether or not a phrase references a phrase preceding the target. If so, it checks whether this would extend the back-reference beyond the window limits. If both conditions have been met, the algorithm appends the phrase to the list E, which stores pointers to phrases with extended references.

The algorithm exits once it reaches a phrase further than w symbols from the first target phrase π_a, or reaches the final phrase of the parsing Π.

This algorithm runs in $O(w \times \texttt{select})$ time and space.

7.3.4 Adjust back-references

The final "helper function" of the LZ-Local editing algorithm adjusts the phrase back-references to account for the edits made to the parsing. This is done in Algorithm 12 and is the equivalent to LZ-End's Algorithm 6.

The function is very similar to its equivalent function for LZ-End. The primary differences are:

- Algorithm 12 is called once for each phrase whose back-reference needs updating. This differs from the LZ-End algorithm which adjusts all

Algorithm 12 `LZ-Local adjust references`(Π, p, D, R, b, e)

Require: LZ-Local parsing Π
Integer p, being the index to the phrase in Π whose back-reference we adjust in this algorithm.
Integer array D, being pointers to dependent phrases in Π.
Integer array R, being the number of phrases replacing each dependent phrase.
Integer b, indexing the first phrase in Π following the target.
Integer e, denoting the difference in number of target phrases after the edit.
Result: The phrase π_p, with an altered back-reference.

1: $abs \leftarrow p - \pi_p.b$
2: $dep \leftarrow D.size - 1$
3: **while** $dep \geq 0$ **and** $abs < d_{dep} + r_{dep}$ **do**
4: $\quad abs \leftarrow abs - r_{dep}$
5: $\quad dep \leftarrow dep - 1$
6: **end while**
7: **if** $abs < b$ **then**
8: $\quad abs \leftarrow abs - e$
9: **end if**
10: $\pi_p.b \leftarrow p - abs$
11: **return** π_p

phrases following the target.

- The LZ-Local back-references are relative (rather than absolute) pointers.
- The LZ-Local algorithm (Algorithm 12) assumes that the target phrases and dependents have already been replaced. This is different to the LZ-End equivalent, which adjusts back-references *in anticipation of the edits being applied.*

We can in linear time compute an array which will store the adjustment for each phrase based on its position and back-reference. This will make each subsequent call to Algorithm 12 run in constant time.

7.3.5 Formal editing algorithm for LZ-Local

Algorithm 13 formalises the editing algorithm for LZ-Local. The parameters of the function are identical to that of LZ-End's editing function, with one additional parameter being the window size used to construct the LZ-Local parsing.

A major difference between the `LZ-Local edit()` algorithm and `LZ-End edit()` is the point at which each algorithm applies the edits to the parsing. The LZ-End variant (Algorithm 8) makes all changes to the parsing Π at the end of the algorithm. This enables Algorithm 5, which re-parses the dependent phrases, to make use of the `rank` and `select` queries without relying on a dynamic structure to support these. `LZ-Local edit()` (Algorithm 13), on the other hand, makes the changes to the parsing as the algorithm progresses. That is, the `LZ-Local edit()` algorithm replaces the target phrases and each dependent phrase as they are identified. This does not affect the implementation of the `rank` or `select` queries, since neither query is required after editing the first phrase.

Before explaining the algorithm, let us recall the edit operation: If the parsing Π represents the raw string T of length n compressed with window size w, then `LZ-Local edit`(Π, w, i, j, S) replaces symbols $t_{i,j-1}$ with S, and updates the parsing Π to reflect this change. After a call to Algorithm 13, Π will therefore decode to $t_{0,i-1} \mid\mid S \mid\mid t_{j,n-1}$.

The `LZ-Local edit`() function starts by identifying the target phrases which will be edited (Line 1), as well as the prefix and suffix of the first and last target phrases, respectively (Line 2). Like the LZ-End variant, the prefix length refers to the number of symbols at the start of π_a which are not part of the target. Similarly, the suffix refers to the number of symbols at the end of the π_b which are not part of the edit.

Lines 3 to 5 identify the phrases which depend on the edited phrases, and those phrases whose back-references will be extended beyond the window limit.

Algorithm 13 `LZ-Local edit`(Π, w, i, j, S):

Require: The LZ-Local parsing Π consisting of m phrases, which forms the LZ-End parsing of a string T of length n.
Integer w, denoting the sliding window size used to construct Π.
Integers i, j, such that $0 \leq i \leq j$.
A string of symbols S.

Result: The array Π which has had symbols $t_{i,j-1}$ removed and S inserted in their place.

$a \leftarrow \texttt{rank}(i)$, $b \leftarrow \texttt{rank}(j)$
$pref\_len \leftarrow \pi_a.l - (\texttt{select}(a) - i)$, $suf\_len \leftarrow \texttt{select}(b) - j + 1$
$D \leftarrow$ `identify dependent phrases`(Π, w, a, b, suf_len) (Algorithm 10)
$E \leftarrow$ `identify extended references`$(\Pi, w, a, b, S.size + i - j)$ (Alg. 11)
$D \leftarrow \texttt{sort}(D \mid\mid E)$
$U \leftarrow$ `random access`$(\Pi, i - pref\_len - w, pref\_len + w) \mid\mid S$
$\mid\mid$ `random access`(Π, j, w) (Algorithm 3)
$a' \leftarrow \texttt{rank}(i - pref\_len - w)$
$b' \leftarrow \texttt{minimum}(m - 1, \texttt{rank}(j + w))$
$\Phi \leftarrow$ `incremental parse`$(U, \pi_{a',a-1}, w, w, pref\_len + S.size)$ (Alg. 9)
if $suf\_len > 0$ **then**
    $\pi_b.l \leftarrow suf\_len - 1$
else
    $b \leftarrow b + 1$
end if
$\Pi \leftarrow \pi_{0,a-1} \mid\mid \Phi \mid\mid \pi_{b,\Pi.size-1}$
$c \leftarrow \Phi.size + a$, $e \leftarrow b - a + \Phi.size$
$p \leftarrow pref\_len + w + S.size$
$R \leftarrow \{0^{D.size}\}$
$dep \leftarrow 0$, $r\_total \leftarrow 0$
$b' \leftarrow b' + e$
while $c < b'$ **and** $dep < D.size$ **do**
    if $d_{dep} + r\_total = c$ **then**
        $\Phi \leftarrow$ `incremental parse`$(U, \pi_{a',c-1}, w, p, \pi_c.l + 1)$ (Algorithm 9)
        $p \leftarrow p + \pi_c.l + 1$
        $\Pi \leftarrow \pi_{0,c-1} \mid\mid \Phi \mid\mid \pi_{c+1,\Pi.size-1}$
        $r_{dep} \leftarrow \Phi.size - 1$, $r\_total \leftarrow r\_total + r_{dep}$
        $c \leftarrow c + r_{dep}$, $b' \leftarrow b' + r_{dep}$
        $dep \leftarrow dep + 1$
    else
        $\pi_c \leftarrow$ `LZ-Local adjust references`(Π, c, D, R, b, e) (Alg. 12)
        $p \leftarrow p + \pi_c.l + 1$
    end if
    $c \leftarrow c + 1$
end while
$finish \leftarrow p + w$
while $c < \Pi.size$ **and** $p < finish$ **do**
    $\pi_c \leftarrow$ `LZ-Local adjust references`(Π, c, D, R, b, e) (Algorithm 12)
    $p \leftarrow \pi_c.l + 1$, $c \leftarrow c + 1$
end while
return Π

These phrases will be treated equivalently in future steps, so the algorithm merges the two arrays and sorts the result.

Line 6 decodes the prefix and suffix of the target phrases, as well as the w symbols which precede and follow the target phrases. If $i < w$, the argument to the random access function will be negative. This does not cause a problem, as the first two lines of the `random access()` function (Algorithm 3) account for this. Into this decoded string we insert the string S.

Lines 7 and 8 identify the phrase pointers a' and b'. These pointers refer to the phrases in Π which parse the first and last decoded symbols of U, respectively.

The algorithm then parses the inserted string S, as well as the prefix to the first target phrase, using the `LZ-Local incremental parse()` function. The array Φ stores the parsing of these symbols (Line 9). Note that the `LZ-Local incremental parse()` function must reconstruct the internal state of the LZ-Local parsing, which is possible to do in no more time than is required to parse the first w symbols of U. The internal state of the parser can be stored for future calls to `LZ-Local incremental parse()`, but we do not show that in this algorithm.

The remainder of the algorithm requires that phrase π_b encodes the first phrase following the edited target. This is done by checking whether or not suf_len is non-zero. If so, the algorithm shortens the length of phrase π_b so that this phrase only encodes the symbols following the edited target (Line 11). Otherwise, we increment b, so that π_b is the first phrase following the target (Line 13).

Line 15 inserts the parsing of the updated target (Φ) in place of the target phrases in Π.

At this point, the parsing of the inserted string S has replaced the target phrases in Π. Two tasks remain:

1. Replacing the parsing of those phrases identified in the array D. Recall that these are the dependent phrases, as well as the phrases whose back-references have been extended beyond the window limit w.
2. Altering the back-reference pointers of phrases following the target, to account for the change in the number of phrases encoding the target and the alternate parsing of the phrases referenced by D.

Lines 16 to 20 initialise the variables necessary to perform these tasks. At each step of the `while` loop in Line 21:

- c points to the current phrase in Π which is being checked.
- e stores the difference between the number of target phrases and the size of the parsing of the inserted string S. This number may be positive or negative.

- p points to the first symbol in the decoded string U which is represented by phrase π_c.
- R corresponds to D, where each entry in R will store the difference between the number of phrases which replace the dependent (or extended) phrase in D and the size of the original dependent (which was size 1).
- *dep* stores the pointer to the next (as yet unprocessed) phrase in D.
- $r\_total > 0$ stores the sum of all elements in R.
- b' gets updated to store the new index in Π of the phrase which encodes the last symbol in our decoded string U.

The `while` loop in Line 21 iterates once for each phrase following the last target phrase (π_b). The loop ends when it has processed phrases encoding more than w symbols.

The conditional in Line 22 asks whether or not the current phrase must be replaced. If yes, the algorithm re-parses the phrase into parsing Φ, which consists of possibly more than one replacement phrase. The algorithm inserts the array Φ in place of the current phrase π_c. Note that this call to the `LZ-Local incremental parse()` function does not require rebuilding the internal state of the LZ-Local parser, if we have stored that state from the initial call to this function at Line 9.

If phrase π_c does not need replacing, the algorithm checks whether its back-reference needs adjusting (Line 29). This uses Algorithm 12. If we are storing the internal state of the LZ-Local parser for future calls to the `LZ-Local incremental parse()` function, we must add the parsing of phrase π_c to this state. Again, the details of this are not included in this thesis.

The loop in Line 21 will terminate either when phrase π_c is further than w symbols from the last target phrase, or when there are no more phrases in need of replacing.

Replacing the phrases in D is not the end of the editing algorithm: Any phrase which references a phrase in or near one of the phrases in D may also need its back-reference adjusted. This is done by the `while` loop starting in Line 36. This performs the same function as the `else` part of the previous `while` loop in Line 29. At this point, we no longer need to store the state of the LZ-Local parser, as the `LZ-Local incremental parse()` function will not be called again.

To complete, the algorithm returns the updated parsing Π.

7.3.5.1 A consideration of the editing algorithm

Given that the editing algorithm decodes the *Window_Size* symbols which precedes and follows the edit location (Algorithm 13, Line 6), one may consider simply editing this decoded string, and reparsing the entire string. However,

this will not result in a valid LZ-Local parsing: the phrases at the end of this "re-parsed" section will be back-referenced by the phrases which encode the following *Window_Size* symbols. Therefore, the phrases in the re-parsed section must have phrase boundaries in the same position as the original parsing. Otherwise, the following phrases will not back-reference the innovation symbol component of a phrase, which is an essential enabler of random access (and thereby editing).

7.4 Evaluating the editing algorithm

This section theoretically and empirically evaluates the LZ-Local parsing algorithm.

From a theoretical perspective, we are interested in how the run-time of the editing algorithm compares to the run-time of decoding, editing, and re-parsing the string. From an empirical perspective, we are interested in how the size, position, and type of edit affects the compression, as well as how these factors are influenced by the entropy of the compressed string we are editing.

All algorithms in this thesis have been implemented and tested rigorously for correctness. We have implemented a naïve version of the `LZ-Local incremental parse`() function (Algorithm 9). However, this implementation has a slower run time than that of LZ-End (Algorithm 2) and the details are uninteresting from a research perspective.

7.4.1 Theoretical evaluation

This section theoretically measures the run-time cost of an `LZ-Local edit()` operation.

The first part of the evaluation reports the cost of an edit operation, ignoring the cost of updating the `rank`/`select` data structure (Section 7.4.1.1). This demonstrates how future optimisations to dynamic structures supporting `rank` and `select` queries will affect the runtime of the `LZ-Local edit()` operation. Then, we discuss one possible implementation of a dynamic structure supporting `rank`/`select` queries (Section 7.4.1.2). Finally, we report the runtime of the compressed edit operation, inclusive of the cost of updating the proposed `rank`/`select` structure (Section 7.4.1.3).

In all stages of the theoretical evaluation, we have compressed a string T of length n into LZ-Local parsing Π of length m, using a window size w. If $w \geq n$, there is no difference between the LZ-Local and LZ-End parsings. Since we have discussed the time complexity of LZ-End edits in Section 6.4, in this section we only consider the case where $w < n$.

To this string, we wish to perform the operation `edit`(i, j, S). Recall from Section 6.1.2 that there are two ways to do this:

1. A raw edit involves decompressing, editing, and recompressing the string.
2. A compressed edit applies the `LZ-Local edit()` function (Algorithm 13) to the parsing Π.

7.4.1.1 Cost of LZ-Local edit, ignoring `rank`/`select`

Let us apply a compressed edit `LZ-Local edit`(Π, w, i, j, S) using Algorithm 13. The runtime cost of this is made up of the following components:

- Identifying the target phrases: $2 \times \texttt{rank} + 2 \times \texttt{select}$ queries.
- Identifying dependent phrases and those with extended back-references: Algorithms 10 and 11 can be combined into one, in which case they collectively run in $O(w \times \texttt{select} + \texttt{rank})$ time. By combining these two algorithms, the resulting array D will not need to be sorted (Algorithm 13, Line 5)
- Decoding the string U: $\Theta((2w + pref\_len + suf\_len) \times (\texttt{rank} + \texttt{select}))$. Note that the LZ-Local parsing means that no phrase can have a length greater than w. Therefore, $pref\_len$ and suf_len are both $< w$, and the runtime function is $\Theta(w \times (\texttt{rank} + \texttt{select}))$.
- Identifying phrases a' and b': $2 \times \texttt{rank}$.
- Parsing the string S, the prefix of π_a, and any phrases in the array D: Parsing the string S and the prefix of π_a will cost $\mathcal{F}(S.size + pref\_len)$, where $\mathcal{F}(\cdot)$ is the cost of constructing the LZ-Local parsing of a string. In order to parse this string, however, the `incremental parse()` function must construct the internal state of the LZ-Local parser. This involves processing the phrases encoding the w symbols preceding the target phrase. This costs no more than $\mathcal{F}(w)$, i.e., parsing a string of length w. All phrases within w symbols following the target need to either be re-parsed, or added to the internal state of the parser. This will cost no more than an additional $\mathcal{F}(w)$.

  Therefore, the worst case occurs when all phrases within w symbols of the target are dependent and need to be replaced, which has a cost of $\mathcal{F}(pref\_len + S.size + 2w)$.
- Finally, adjusting the references of the phrases encoding $2w$ symbols following the target. With a pre-computation costing $O(w)$, we can adjust each phrase's back-reference in constant time (Section 7.3.4).

The time complexity of the compressed edit is therefore:

$$\Theta(w(\texttt{rank} + \texttt{select})) + \mathcal{F}(pref\_len + 2w + S.size) \tag{7.1}$$

Note that the `rank` and `select` operations are all performed prior to making any changes to the parsing. Therefore, if the data structure supporting these queries means that the run-time of either operation is dependent on the size of the parsing Π, then this is related to the parsing size *prior to applying the edit.*

Recall that Equation (7.1) does not include the cost of updating the data structure supporting `rank`/`select` queries. The next section addresses this task.

7.4.1.2 Cost of the `rank` and `select` operations

We have not yet addressed the cost of the `rank` and `select` queries, or the cost of maintaining the associated data structure which makes these queries possible. In the LZ-End case, we decoupled the maintenance of the data structure from the editing algorithm (recall Section 6.5). We accepted that the best solution may involve rebuilding the data structure from scratch; however, we could ignore this, since the same cost would apply to the raw edit.

Unfortunately, we cannot do the same in the case of LZ-Local without consequences: We have developed a method of *locally* editing the data. If this edit involves rebuilding a data structure that covers the whole parsing, we will lose the local bounds of the editing algorithm.

Fortunately, data structures like that developed by Pătraşcu and Thorup [79] can address this issue: being based on fusion trees, their data structure can resolve `rank` and `select` queries, *as well as insert into or delete entries from* the data structure. All of these operations can be completed in $O\left(\frac{\log m}{\log w}\right)$ time and space [79]. Here, m is the number of integers stored in the structure (i.e., the size of the parsing Π), and $\log w$ is the maximum size of the integers in the structure. This structure will store phrase lengths, which can not exceed the window size in LZ-Local parsing.

This brings us to the question: How many times does the `rank`/`select` data structure need to be updated? This depends on:

- The number of phrases which the edit operation removes: $b - a$.
- The number of phrases which the edit operation inserts: $\Phi.size$.
- The number of dependent phrases, and their replacements:

$$R.size + \sum_{i=0}^{i<R.size} r_i$$

where each variable is taken from Algorithm 13.

Remark 7.6. Note that the LZ-Local edit function inserts into a static array in multiple places (Algorithm 13, Lines 15 and 25). Ordinarily, the cost of each insertion would be linear with respect to the number of phrases inserted, and the number of phrases which follow the location of the insertion. However, we have not counted the cost of editing the static array of phrases. The cost of editing the LZ-Local parsing will be dependent whatever structure supports the `rank` and `select` queries.

We leave identifying and evaluating this structure as future work.

7.4.1.3 Cost of LZ-Local edit

Theorem 7.2. The cost of the LZ-Local editing algorithm is:

$$O\left(\frac{w \log \bar{m}}{\log w}\right) + \mathcal{F}(pref\_len + 2w + S.size)$$

where $\bar{m} = \texttt{maximum}(\Pi.size, \Pi'_e.size)$, $pref\_len < w$ and $\mathcal{F}(x)$ is the cost function of parsing a string of length x. Recall that $\Pi.size$ is the size of the parsing prior to the edit being applied, $\Pi'_e.size$ is the size of the parsing following the edit.

Proof. Equation (7.1) shows us that the cost function without editing the `rank`/`select` data structure is

$$O(w(\texttt{rank} + \texttt{select})) + \mathcal{F}(pref\_len + 2w + S.size)$$

We therefore need to calculate:

(a) The cost of the `rank` and `select` queries,
(b) the cost of updating the `rank`/`select` data structure, and
(c) the number of edits we need to make to the `rank`/`select` structure.

We know that the cost of each `rank` or `select` query is $O\left(\frac{\log x}{\log w}\right)$, where the structure contains x integers, each no larger than w. Each insertion into or deletion from the structure also costs the same as a query [79].

We also know that there are w symbols which could depend on the target phrase: therefore, there are at most w phrases which could depend on the target phrases, and at most w phrases which could replace the dependent phrases. Therefore, $O(w)$ edits need to be made to the structure to edit the dependent phrases.

The only question we now need to answer is: How large is x, the number of integers in the structure? This is equal to the size of the parsing Π. However, the size of Π changes as we make edits to it. Therefore, we set a variable

$\bar{m} \leftarrow \texttt{maximum}(\Pi.size, \Pi'_e.size)$, where Π is the parsing prior to a call to the `LZ-Local edit()` function, and Π'_e is the parsing following the compressed edit.

Therefore, the cost of the edit becomes

$$O\left(w\left(\frac{\log \bar{m}}{\log w} + \frac{\log \bar{m}}{\log w}\right)\right) + \mathcal{F}(pref\_len + 2w + S.size) + O\left(w \times \frac{\log \bar{m}}{\log w}\right)$$

which simplifies to the result. □

7.4.1.4 Comparing compressed editing time to raw edit

Let us now consider the case where we apply $\texttt{edit}(i, j, S)$ as a raw edit:

- Decompressing the string costs $\Theta(n)$ time,
- editing the raw string costs $\Theta(j - i + S.size)$,
- re-compressing will cost some function $\mathcal{F}(n')$, where $\mathcal{F}()$ is the cost function of constructing the LZ-Local parsing, and $n' = n-(j-i)+S.size$ is the new length of the edited string, and finally,
- constructing the `rank`/`select` data structure will cost some function $\mathcal{G}(\cdot)$. The parameter to this function will depend on the properties of the data structure: e.g., the indexable dictionary used in the original LZ-End parsing function [2, 61] was constructed over a bit-vector of equal length to the raw string (n'). However, other indexable dictionaries may be constructed over the parsing Π, in which case the argument to this cost function will be $\Pi'.size$, i.e., the size of the new parsing.

In total then, the cost of the raw edit will be:

$$\Theta(n + j - i + S.size) + \mathcal{F}(n') + \mathcal{G}(\cdot) \tag{7.2}$$

Recall that by definition, $j - i < n$.

Note that in this case, the `rank`/`select` structure can be static; it need not be the same dynamic structure required by the compressed edits. Therefore, we can assume that the `rank` and `select` queries are completed in constant time.

It is not possible to make a direct comparison between this function and that of the compressed edit (Theorem 7.2). Future work will need to investigate dynamic data structures which answer `rank` and `select` queries, and allow updates, in time that does is not dependent on the size of the structure. This will allow us to properly compare the compressed and raw editing functions, and define the circumstances in which each will be faster than the other.

7.4.1.5 Locality of editing

Applying `LZ-Local edit`(Π, w, i, j, S) will potentially alter the parsing of phrases which encode symbols $T_{i,j+w}$. In addition, the back-references of phrases which encode symbols $T_{j+w,j+2w}$ may need to be updated (Algorithm 13, Line 37). Finally, the phrases which encode symbols $T_{i-w,i}$ will need to be decoded (Line 6). Recall that each phrase is at most w symbols from its back-reference; however, this back-reference is again at most w symbols from it's back-reference. Also recall that the length of a phrase is at most w (Definition 7.1). Therefore, in the worst case, we would need to read the phrases which encode symbols in the range $T_{i-w^2,i}$.

This is an improvement on the LZ-End Edit algorithm, which can edit phrases at any point following the location of the edit.

7.4.2 Empirical evaluation

To empirically evaluate the LZ-Local editing algorithm, we measure the *modification ratio* (MR):

$$\text{MR} = \frac{\Pi'_e.size}{\Pi'.size}$$

where $\Pi'.size$ is the number of compressed phrases achieved by compressing the string following a raw edit. $\Pi'_e.size$ is the number of phrases when edit e has been applied as a compressed edit. A modification ratio of 1 will mean that the two methods compress to the same number of phrases. Any value larger than 1 represents the loss of compression introduced by the compressed edit.

Recall that an edit can be of three forms: string insertion, string deletion, or string replacement. When referring to the MR of a specific type of edit, we use the terms *insertion ratio, deletion ratio*, or *replacement ratio*, respectively.

To keep things simple, we do not evaluate the compressed edit against all 10 entropy calibrations used in Section 7.2. Rather, we assess only against the three calibrations $\epsilon \in \{0.00025, 0.0025, 0.025\}$. The compression ratios reported in Section 7.2 showed consistent results for the different entropy calibrations: We are therefore confident that the effect of entropy on the different compression measurements will be highlighted by these three calibrations. Note that these three entropy calibrations represent compression ratios between 0.2 and 0.9 for LZ-Local; therefore, the results will cover strings achieving low, medium and high compression ratios.

We also do not evaluate against all sliding window sizes reported in Section 7.2. Rather, we only consider window sizes $w \in \{4096, 16384, 65536\}$. This is necessary due to computational restrictions, and the need to limit the complexity of the resulting graphs.

Remark 7.7. There are a number of reasons for choosing to evaluate the editing algorithm by comparing phrase counts, rather than compression ratios (which take into account the size of each phrase as well as the phrase count):

- The size of the phrases might change during an edit:
  Changing the back-references of dependent phrases will sometimes change the number of bits required to encode the back-references. These changes in phrase size can have a large impact on the compression ratio, without having an impact on the phrase count. Since the phrase count was the original measure of LZ-complexity [1], it is a more meaningful metric than outright compression ratio.
- Any metric reporting the compressed size should include the `rank`/`select` data structure.
  The editing algorithm applies regardless of the choice of `rank`/`select` data structure, so the experiments should also generalise to the use of any data structure supporting `rank`/`select` queries.
- The size of the data structure supporting `rank`/`select` queries may be dependent on the phrase count. Therefore, reporting phrase counts will allow generalising the results to any future data structures or representations of the LZ-Local parsing. This generalisation would not possible from the compression ratio alone.
- Section 7.2.2.1 showed that the LZ-77 and LZ-Local phrase counts are very consistent, while the compression ratio has a much higher variance, which is predominantly impacted by the phrase size.

The next section examines how the position of the edit affects the resulting compression, and the following section considers the impact of different size edits. Finally, we conclude by comparing the impact of many small edits on the MR.

7.4.3 The position of the edit

This section evaluates the effect which changing the position of the edit has on the compression ratio. To do this, we make edits of size 100 symbols, in varying positions within the calibrated strings. This means that the edit is $\approx 2 \times 10^{-5} \times l$, where $l = 5 \times 2^{20}$ is the length of the string (in bytes) which we are editing.

We make the edits at positions $\frac{i \times l}{10}$, where l is the length of the raw string, and $0 \leq i \leq 9$. We do not make edits at the end of the string; in that case, the edited phrases will have no dependents, and the MR will be exactly 1.

Figures 7.36 to 7.38 show the effect which changing the position of an edit has on the MR.

The graphs show that the position of an insertion (Figures 7.36a, 7.37a and 7.38a) has minimal impact on the MR. This is true for both LZ-End and the three window sizes for LZ-Local, and for strings of low, medium and high entropy. In all cases, the MR is very near to 1, and is often less than 1, regardless of the position of the insertion. The only noticeable effect which the position of the insertion has on MR is to reduce the variance in observed MR's as the insertion is applied closer to the end of the string. This effect is noticeable in the medium and high entropy strings (Figures 7.37a and 7.38a).

The position of a replacement has a nearly identical impact on MR as for a deletion. This is because a replacement can be viewed as a deletion and an insertion applied in concert. Since the insertion has such a minimal impact on MR, and the deletion has a larger impact (at least for those deletions at the start of the string), the impact of the deletion dwarfs the impact of the insertion. The result is that changing the position of the edit has the same impact on MR for replacements as for deletions.

If the edit is applied to the start of the string, replacements and deletions both have a much larger detrimental effect on the LZ-End MR than the equivalent edit has on the LZ-Local MR. The effect of this is larger for lower entropy strings. This effect rapidly disappears as the edit is applied further from the start of the string.

For all window sizes of LZ-Local, the position of the replacement or deletion has little impact on the MR. In all cases, the MR is very close to 1, and the variance decreases as the edit is applied closer to the end of the string.

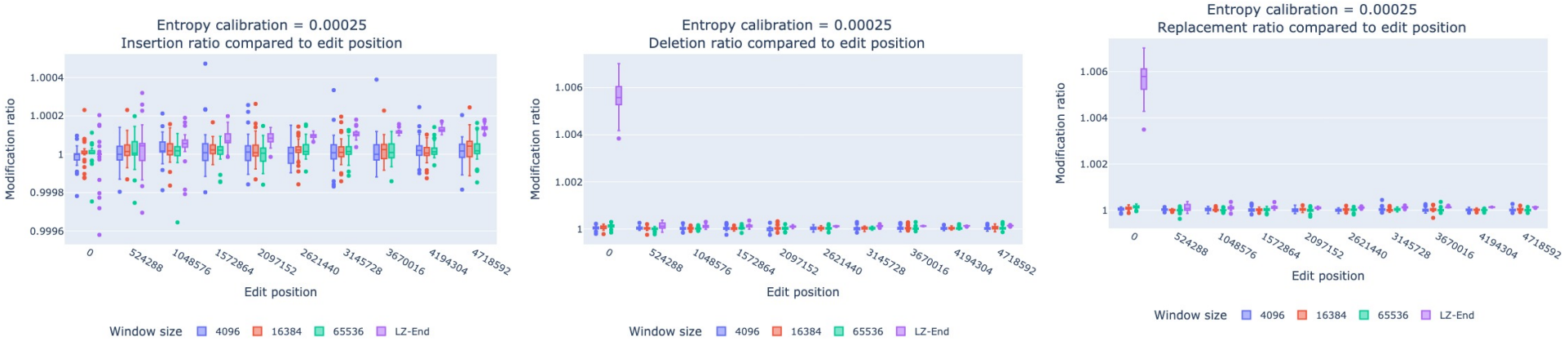
7.4.4 Size of the edit

This section answers the question of how the size of an edit affects the LZ-Local and LZ-End compression.

We make edits of sizes 2^i, where $0 \leq i \leq 19$. Since each string is of size 5×2^{20} prior to the edit being applied, this represents edits up to 10% of the raw string size.

Each edit is made at the midpoint of the string:

- Insertions:
$$\texttt{edit}\left(\Pi, \frac{\texttt{raw size}}{2}, \frac{\texttt{raw size}}{2}, \texttt{string}\right)$$
- Deletions:
$$\texttt{edit}\left(\Pi, \frac{\texttt{raw size} - \texttt{edit size}}{2}, \frac{\texttt{raw size} + \texttt{edit size}}{2}, \emptyset\right)$$

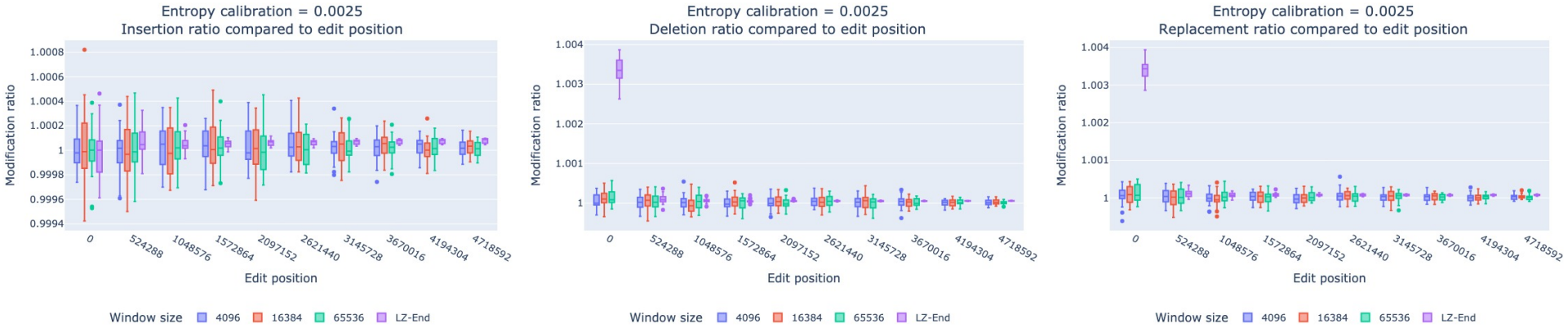


(a) Insertion

(b) Deletion

(c) Replacement

Figure 7.36: The effect of position on MR for the different types of edits, on low entropy strings.
The position of the insertion has no noticeable effect on the MR – the overall effect of insertions in any position is minimal.
Replacements and deletions have near identical effects: For LZ-Local, the position has very minimal effect, whereas for LZ-End, there is a comparatively large impact if the edit is applied at the start of the string. This effect quickly disappears as the edit is made further into the string.

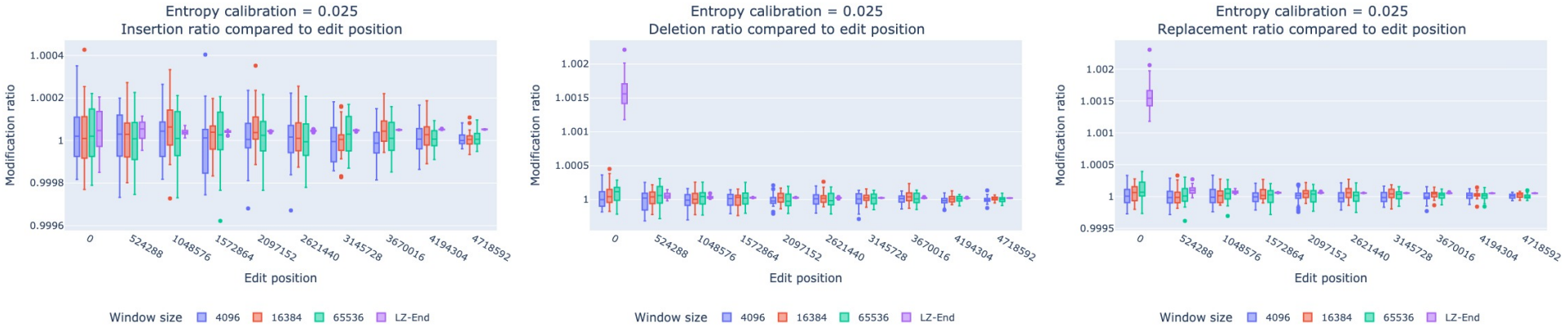


(a) Insertion

(b) Deletion

(c) Replacement

Figure 7.37: The effect of position on MR for the different types of edits, on medium entropy strings.
These graphs show the same results as for the low entropy strings, whereby position of the insertion has a minimal impact, and the position of replacements and deletions have small impacts on the LZ-Local compressed strings. For LZ-End, the negative impact of a replacement or deletion at the start of the string is reduced compared to the low entropy case.
For all positions and parsings, the variance in MR decreases as the edit is made closer to the end of the string.



(a) Insertion

(b) Deletion

(c) Replacement

Figure 7.38: The effect of position on MR for the different types of edits, on high entropy strings.
Note the continuation of the previous trend, in which the position has minimal effect on the insertion ratio, other than that the variance of the insertion ratio decreases as the insertion is made closer to the end of the string. The replacement ratio again mirrors the deletion ratio, and a deletion or replacement at the start of the string has less of a detrimental impact on MR for these high entropy strings than for the medium and low entropy strings.

- Replacements:

$$\texttt{edit}\left(\Pi, \frac{\texttt{raw size} - \texttt{edit size}}{2}, \frac{\texttt{raw size} + \texttt{edit size}}{2}, \texttt{string}\right)$$

Insertion

Figures 7.39 to 7.41 show the effect of different size insertions on the MR. This is done for LZ-End and LZ-Local, when the insertion is applied to low, medium and high entropy strings.

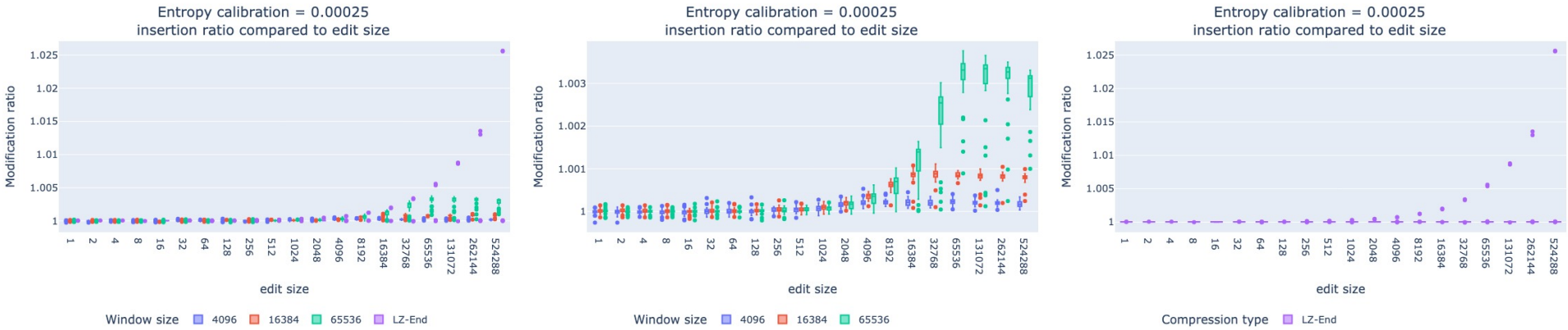
For LZ-Local, we observe that:

- The insertion ratio increases with the size of the insertion. However, the impact of this effect plateaus after the insertion size exceeds the window size.
- Insertions into low entropy strings have a larger modification ratio than those into high entropy strings.

For LZ-End, the experimental results are less conclusive: Note that for the low-entropy insertion (Figure 7.39c), the MR seems unaffected by the insertion size, except for two outliers, where the MR is hugely influenced by the insertion size. Compare this to the medium-entropy insertion (Figure 7.40c), which has a very high variance in MR as the insertion size increases, and to the high-entropy case (Figure 7.41c), for which the MR is very highly dependent on the insertion size.

The appendix (Appendix: A note on alphabets) explains these confusing observations. The key takeaways from this are:

- One must be careful when the source of the compressed string uses a different alphabet to that used by the LZ parser. In our experiments, the sources of our strings used a binary alphabet, whereas the LZ parsers interpreted these strings as bytes.
- The MR for insertions into LZ-End-compressed strings is dependent on the size of the insertion, if the inserted string comes from the same source as the string we are inserting into. We cannot quantify this exact relationship for low and medium entropy strings: future work is required for this.
- The MR for insertions into LZ-End-compressed strings, when the inserted string comes from a different source to the string we are inserting into is likely to be very close to 1. This seems to be true regardless of the size of the inserted string.



(a) (b) (c)

Figure 7.39: The effect of different sizes of insertions on the modification ratio, when inserting into the lowest entropy calibration.
On the left, we show the insertion ratio of LZ-End and three types of LZ-Local. The middle graph isolates the LZ-Local insertion ratios. Finally, the rightmost graph isolates the results for LZ-End.
For the three window sizes of LZ-Local, the MR increases with the size of the edit, but then plateaus. This increase is greater for the larger window sizes. The MR plateaus when the size of the edit reaches and exceeds the window size.
Note that the range of LZ-End's box-and-whisker plots remains only slightly larger than 1.0, except for two outlying strings for which the MR increases significantly with the edit size. This is explained in Appendix: A note on alphabets.

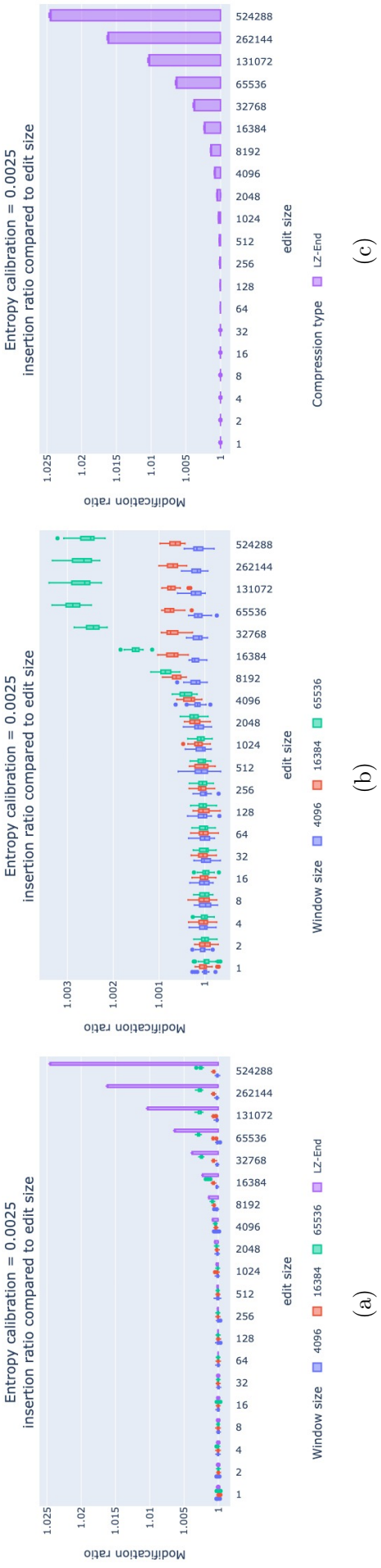
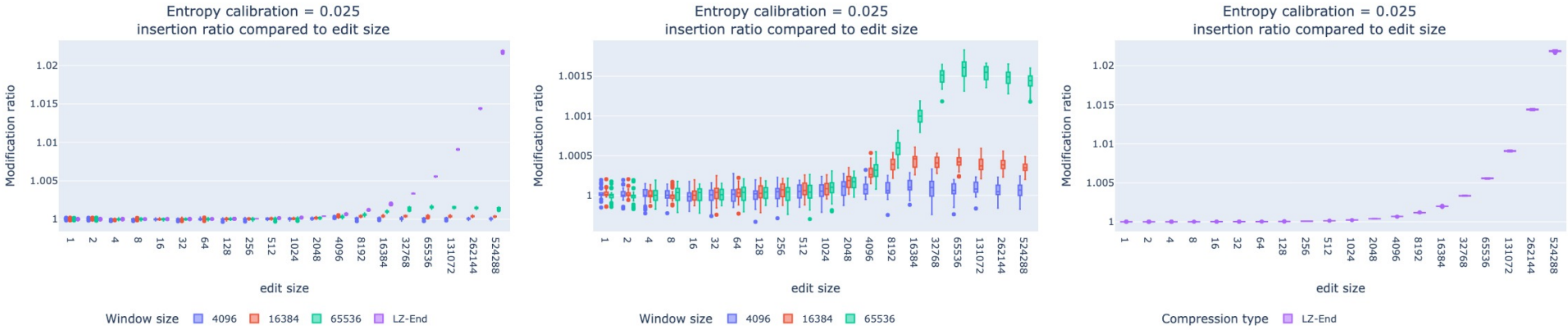


Figure 7.40: The effect of different sizes of insertions on the modification ratio, when inserting into the medium entropy calibration strings. For the three LZ-Local window sizes, note the same increase and plateau in the MR as noted previously for the lower entropy calibration. For LZ-End, more strings display the significant increase in MR than the lower entropy calibration.



(a) (b) (c)

Figure 7.41: The effect of different sizes of insertions on the modification ratio, when inserting into the highest entropy calibration strings.
For the three LZ-Local window sizes, again note the increase and plateau in the MR.
For LZ-End, all strings display the significant increase in MR.

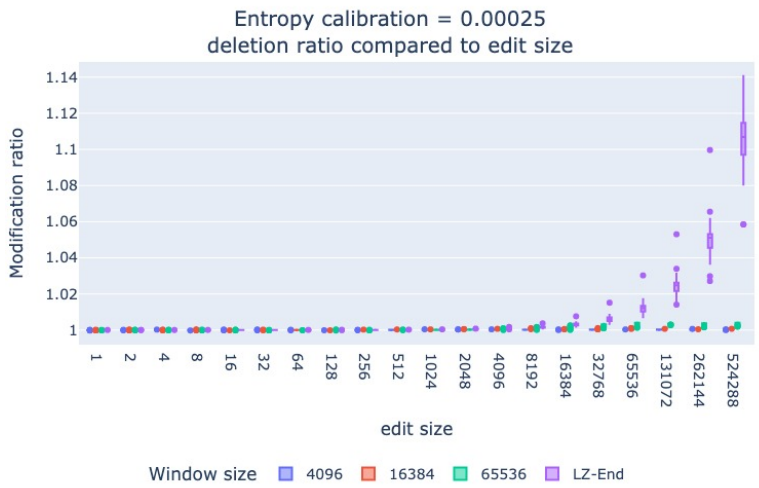
Deletion and Replacement

Figures 7.42 to 7.44 show the effect of different size deletions and replacements on the MR, for low, medium and high entropy strings.

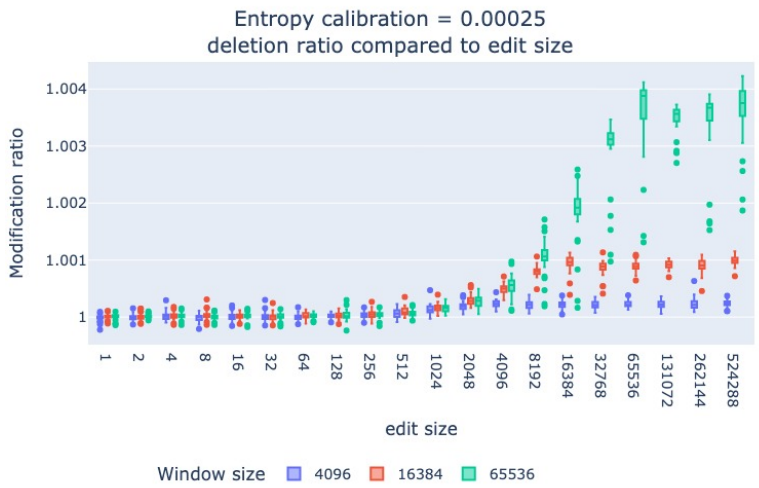
For LZ-Local, we continue to see the MR plateau after the size of the edit has exceeded the window size. For all entropy levels, the MR for LZ-Local is only very slightly larger than 1; however, the MR does slightly decrease as the entropy of the strings increases.

For LZ-End, the shape of the replacement ratio curves generally reflects that of the deletion ratio. There are some differences, however, and these are correlated to the related insertion ratio. For example, consider the medium entropy string: the variance in the LZ-End replacement ratio is much higher for large replacements than the corresponding variance in the deletion ratio (Figures 7.43a and 7.43c). This is because the variance of the corresponding insertion ratio is much higher (Figure 7.40c).

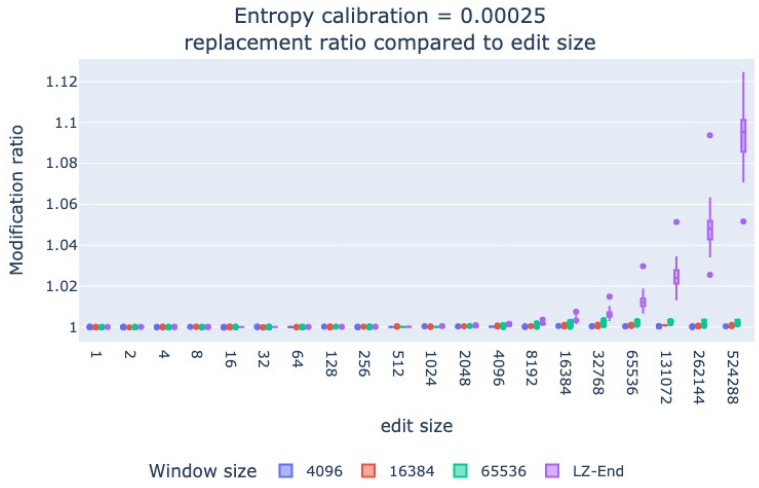
As another example, consider the high entropy string: the variance in the replacement ratio is similar to that of the deletion ratio (Figures 7.44a and 7.44c). However, the difference is that the replacement ratio is notably higher for large edits. This agrees with what we have seen in the corresponding insertion ratio (Figure 7.41c): the insertion ratio for large edits was > 1.02, but with low variance.



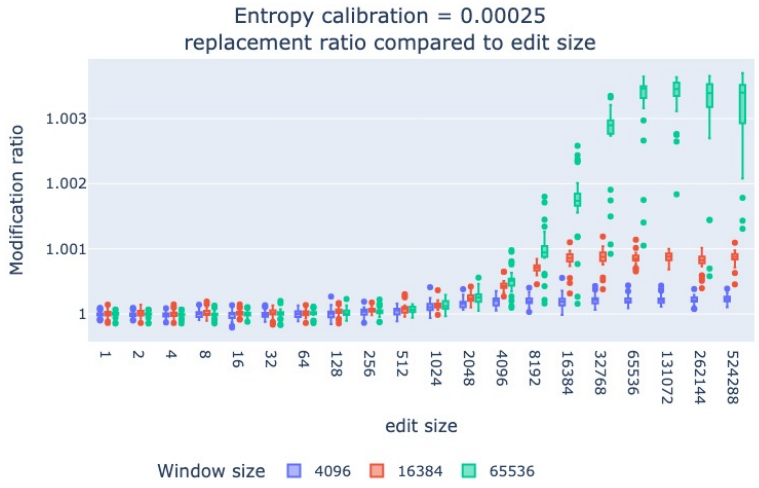
(a) Deletion: Comparing LZ-End with LZ-Local.



(b) Deletion: Zooming in on the LZ-Local graphs.

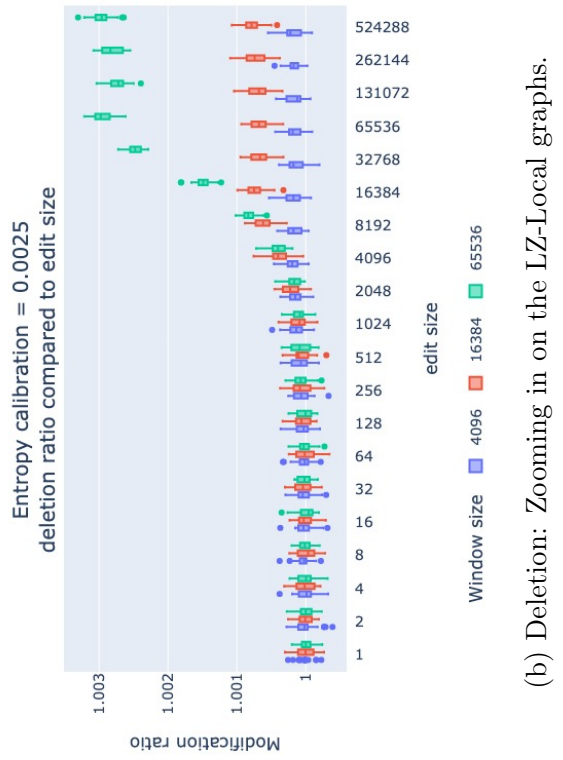


(c) Replacement: Comparing LZ-End with LZ-Local.

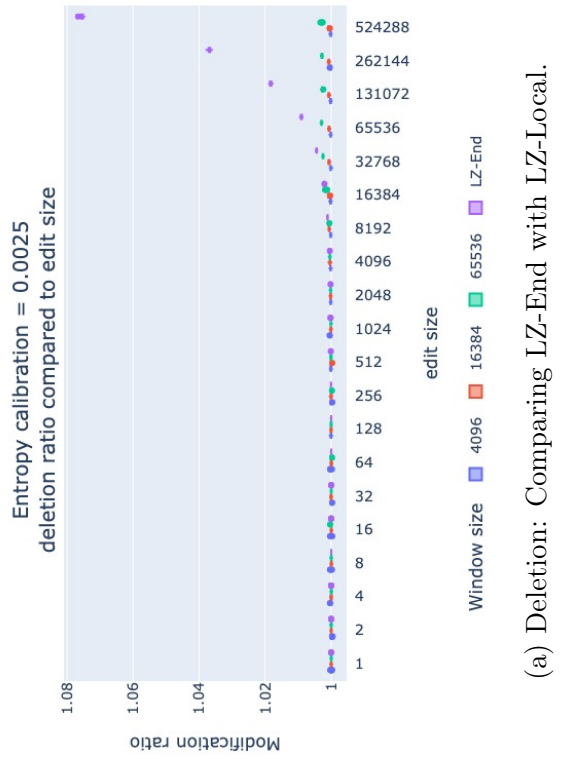


(d) Replacement: Zooming in on the LZ-Local graphs.

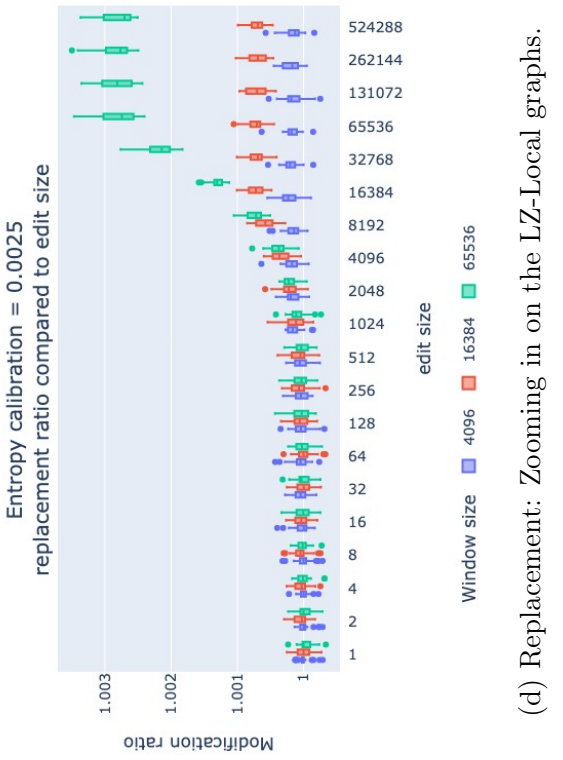
Figure 7.42: Effect of different size deletions (top) and replacements (bottom) on the low entropy strings.
The large edits have a huge effect on LZ-End MR.
For LZ-Local, however, the MR reaches a plateau after the size of the replacement exceeds the window size (see the right-hand figures).



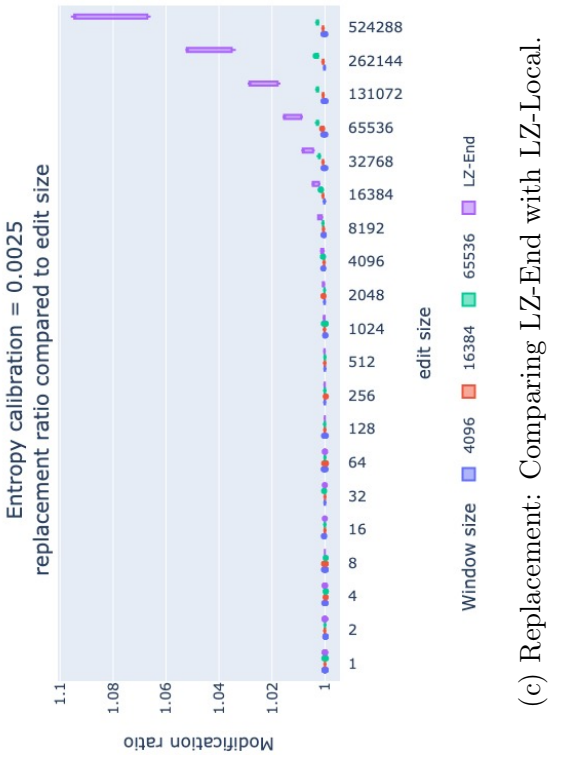
(b) Deletion: Zooming in on the LZ-Local graphs.



(a) Deletion: Comparing LZ-End with LZ-Local.

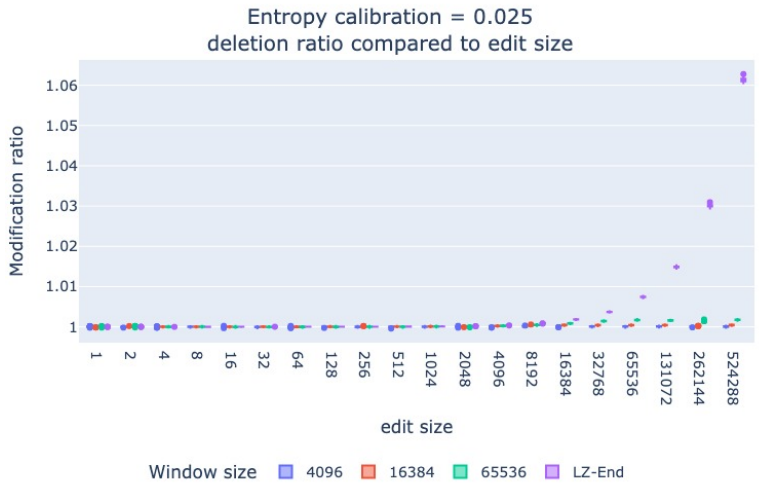


(d) Replacement: Zooming in on the LZ-Local graphs.

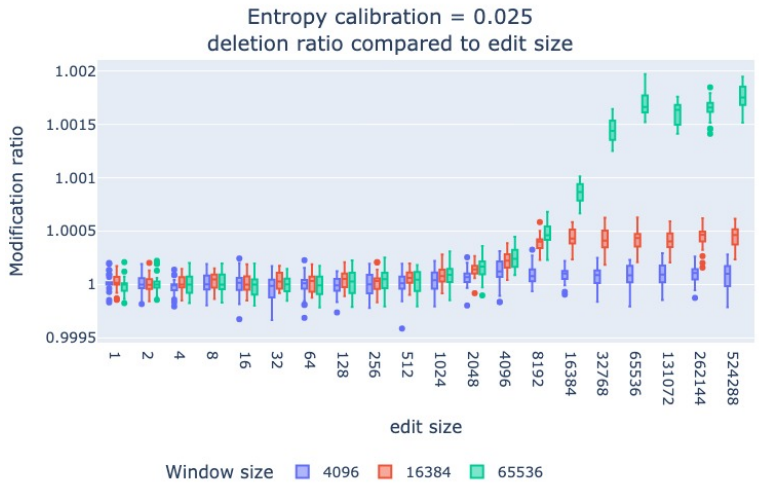


(c) Replacement: Comparing LZ-End with LZ-Local.

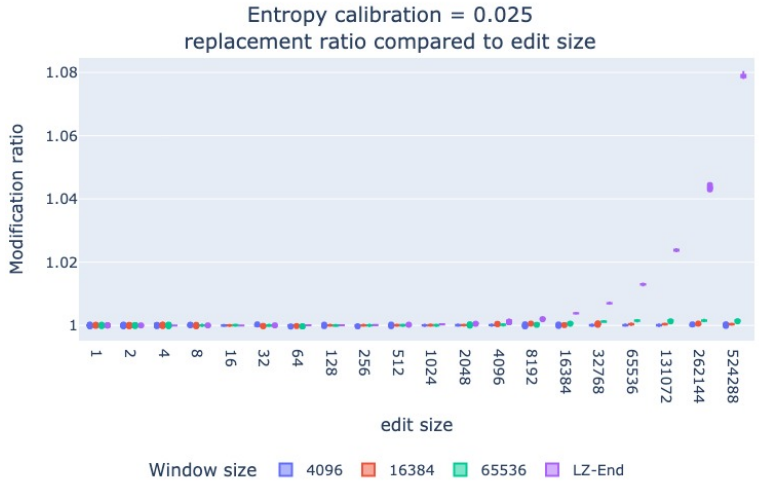
Figure 7.43: Effect of different size deletions (top) and replacements (bottom) on the MR for the medium entropy strings. The result is the same as for the lower entropy case, albeit with lower magnitude increases in MR (for both LZ-End and LZ-Local).



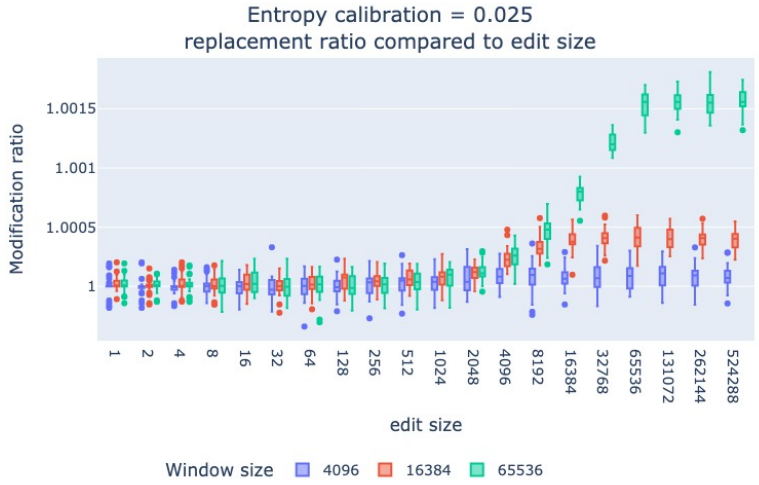
(a) Deletion: Comparing LZ-End with LZ-Local.



(b) Deletion: Zooming in on the LZ-Local graphs.



(c) Replacement: Comparing LZ-End with LZ-Local.



(d) Replacement: Zooming in on the LZ-Local graphs.

Figure 7.44: Effect of different size deletions (top) and replacements (bottom) on the MR for the high entropy strings.
The result is the same as for the low and medium entropy strings: again, the overall effect is of a lower magnitude for these higher entropy strings.
The variance of the results has reduced significantly compared to the same operation on the lower entropy strings.

7.4.5 Incremental edits

This section considers the impact on MR of making many small edits to the compressed string. To assess this, we consider two cases:

(a) Measuring the MR after applying a single edit of which affects 1000 symbols. We call this a *bulk edit*.

(b) Measuring the MR after applying 1000 edits, which each affect a single symbol. We refer to this as an *incremental edit*.

We make these edits at the midpoint of each compressed string.

Note that the edited parsings from each of the above cases will decode to the same raw string. The comparision between bulk and incremental edits, for strings of low, medium and high entropy, is shown in Figures 7.45 to 7.47.

Before describing the results, let us consider the two ways in which a compressed edit is different from the true parsing of the edited string:

- The inserted string is not parsed in the same way as it would be in the true parsing of the edited string.
- The dependent phrases (and in the case of LZ-Local, the phrases with extended back-references) are replaced. These replacement phrases do not necessarily match their true parsing in the edited string.

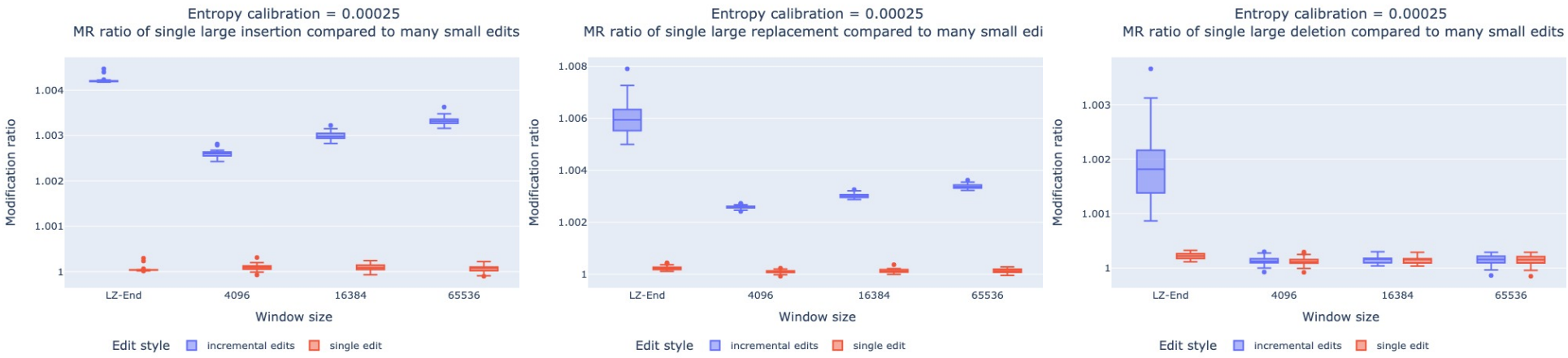
The differences in MR between bulk and incremental edits will be explained by the differences between how the bulk and incremental edits handle these two points.

In the case of a deletion, only the final point applies, as there is no inserted string to parse. In the case of insertions or replacements, both points apply.

The first notable result from the experiments is that LZ-Local deletions result in identical parsings for bulk and incremental edits; we see this in Figures 7.45c, 7.46c and 7.47c. This should not be surprising, as the bulk and incremental deletions will both collectively have the same dependent phrases. The bulk and incremental edits will replace these dependent phrases in the same way, since LZ-Local decodes a dependent phrase and re-parses it in its entirety (see Algorithm 13, line 23).

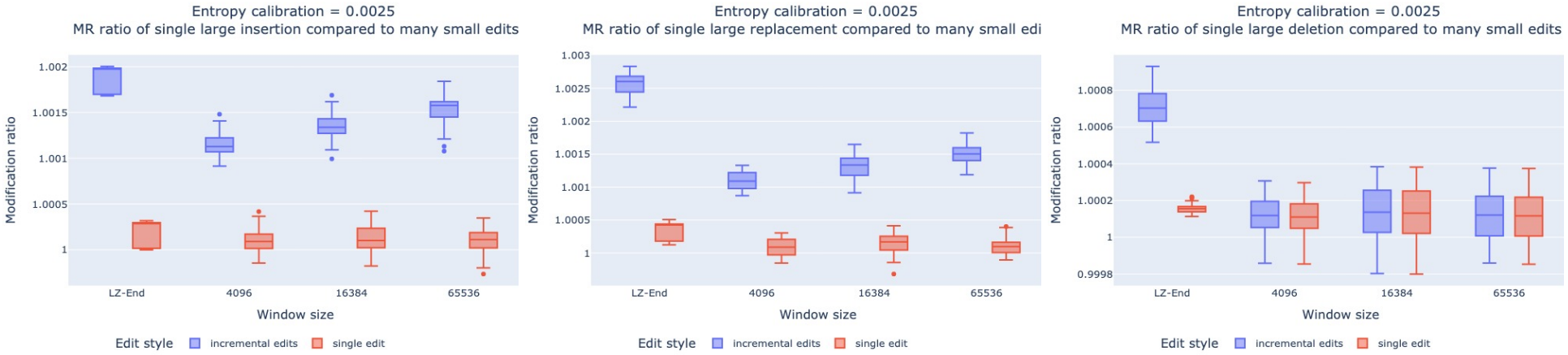
This begs the question: Why is there a difference between incremental and bulk deletions for LZ-End? Recall that the LZ-End editing function replaces dependent phrases by treating the dependent phrase as consisting of three components (see Figure 6.1, repeated for reference). The first component of a dependent back-reference is the *prefix*, which are the phrases preceding the target. The second component is a like-for-like copy of each dependent phrase. Finally, the third component is the *suffix* which are the phrases following the target phrases. This process is formalised in Algorithm 5.

The bulk and incremental MR's would ordinarily be identical for the LZ-End deletions, as they are for LZ-Local. However, the implementation of



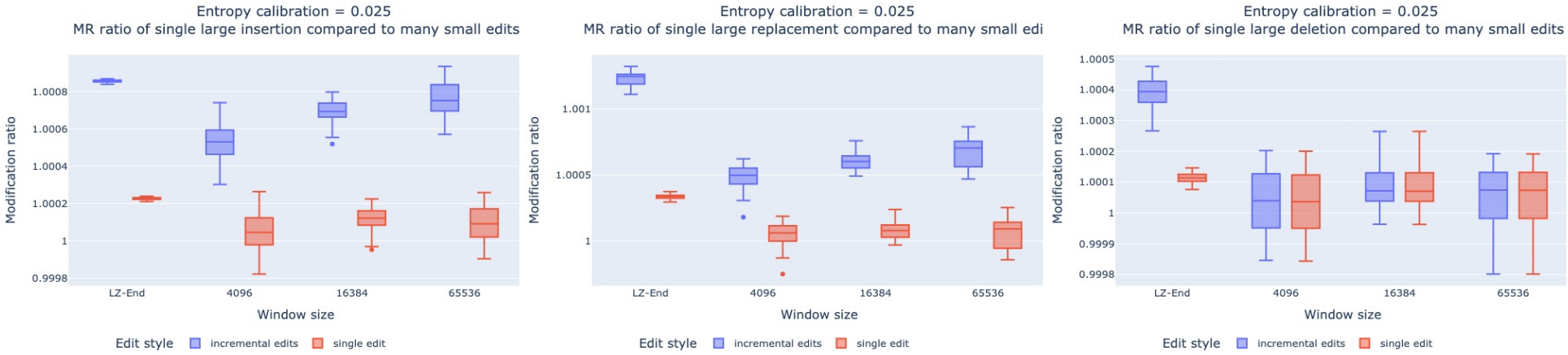
(a) (b) (c)

Figure 7.45: The effect of one bulk versus many incremental edits on the compressed size, when editing low entropy strings.
LZ-Local deletions (7.45c) result in identical parsings, regardless of whether the deletion is applied in bulk or as a series of small deletions. For LZ-End, however, a single large deletion results in a more concise parsing than than many small deletions.
A single large insertion results in a more concise parsing than many incremental insertions; this is true for both LZ-Local and LZ-End (7.45a). A large replacement is similarly more concise than many small replacements, for both LZ-Local and LZ-End (7.45b). Since we know that the LZ-Local deletion ratio is identical for bulk and incremental edits, we conclude that the negative impact which incremental replacements have on the MR is a result of parsing the inserted string, rather than replacing the dependent phrases.



(a) (b) (c)

Figure 7.46: The effect of one bulk versus many incremental edits on the compressed size, when editing medium entropy strings.
We see a repeat of the results from low entropy strings: LZ-Local deletions are identical, regardless of whether they are applied incrementally or in bulk. In all other cases, the single bulk edit is more concise than the smaller, incremental edits.
Note that for the medium entropy strings, there is a smaller difference between the bulk and incremental edits than there is for the low entropy strings in Figure 7.45.



(a) (b) (c)

Figure 7.47: The effect of one bulk versus many incremental edits on the compressed size, when editing high entropy strings.
Again, we see a repeat of the results from the low-and medium entropy strings: LZ-Local deletions are identical, regardless of whether they are applied incrementally or in bulk. In all other cases, the single bulk edit is more concise than the smaller, incremental edits.
Note that for these high entropy strings, there is again a smaller difference between the bulk and incremental edits than there is for the low and medium entropy strings in Figures 7.45 and 7.46.

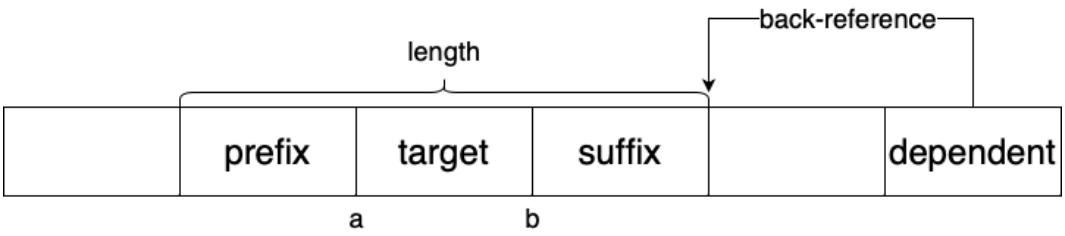


Figure 6.1: The possible components of a dependent phrase's back-reference. The dependent phrase may reference one or more prefix phrases which precede the target (labelled 'prefix'), one or more phrases in the target, and one or more suffix phrases which follow the target (labelled 'suffix').

the experiments used an "optimisation" whereby the replacement of the *suffix* component is greedy. Recall that the suffix of a dependent phrase is encoded by the original dependent phrase, with a shortened length component (see Algorithm 5, line 22). If the target is a single symbol, and this symbol is the first symbol of the phrase encoding it, the implementation of Algorithm 5 replaces the dependent phrase with two phrases: the first encodes only the replaced target symbol (and has length 0), and the second is the original phrase, with a decremented l component.

When incrementally deleting many symbols, this case can be triggered many times: Consider performing the operation `LZ-End edit`$(\Pi, i, i+2, \emptyset)$ as two incremental deletions: `LZ-End edit`$(\Pi, i, i+1, \emptyset)$ followed immediately by `LZ-End edit`$(\Pi, i, i+1, \emptyset)$. When performing the first of the incremental edits, there may be some set of dependent phrases which will have a *suffix* component. Each suffix of the first edit's dependent phrases will also be a dependent of the second operation. These will again be replaced by two phrases: a single phrase $\langle 0, 0, t_i \rangle$, and the suffix phrase with a decremented length component.

The bulk operation, which removes both symbols i and $i+1$ in the same operation, will not have this issue, and the resulting parsing will be more concise than the two incremental edits. This insight highlights the fact that optimisations to the algorithms must be carefully thought through and evaluated.

Another notable result from the experiments is that bulk insertions have a more concise MR than incremental insertions. This is true for LZ-End and LZ-Local, for all window sizes and entropy calibrations (Figures 7.45a, 7.46a and 7.47a). This is due to the parsing of the inserted string: When inserting a single symbol into the LZ-End parsing, that symbol will be parsed as a single phrase with a zero back-reference and length component. When inserting a single symbol into the LZ-Local parsing, we decode the phrase containing the target, insert the symbol into this decoded string, and re-parse it. Unsurprisingly, both the LZ-End and LZ-Local parsings are less concise

when performing incremental insertions than when performing insertions in bulk.

Finally, replacements are a combination of the factors affecting the insertion and deletion ratios. In the case of LZ-Local, where the bulk and incremental deletion ratios are identical, we can conclude that the difference between the bulk and incremental replacement ratios is due to differences in parsing the replacement string, rather than differences in replacing dependent phrases.

7.4.6 Quality assurance for the empirical results

Extensive work has gone into ensuring that the experimental results reported in the empirical evaluation are correct. A comprehensive suite of unit tests consisting of 1,300 assertion statements verified each component of the `edit` functions for both LZ-End and LZ-Local. In addition, tests were built into every step of the experiments themselves, to confirm that every reported result is in fact correct:

- The tests applied each `edit()` operation to the raw string, as well as to the compressed string. The test then decoded the (edited) compressed string, and compared the result to the raw (edited) string. In every case, the two results were identical.
- Following each `LZ-Local edit()` operation (Algorithm 13), the tests involved parsing the entire compressed string to ensure that no phrase's back-reference extended beyond the limits allowed by the *Window_Size*.

7.4.7 Summary of evaluation

The key takeaways from the empirical evaluation are:

- **The sliding window in LZ-Local effectively curbs the negative impact which an edit has on the MR.**
  This is an important point. For LZ-End, the impact of an edit on the MR is at times significantly larger than the size of the edit itself. Consider for example the deletion at the start of a low-entropy string (Figure 7.36b). Deleting 100 bytes from the raw text increased the size of the compressed data by $\approx 5,000$ bytes! For the sliding window cases, the same deletion increased the size of the compressed string by < 350 bytes.
  For LZ-Local, the MR is related to the size of an edit only while the edit is smaller than the window size. Once the edit exceeds the size of the window, there are no additional negative impacts of making a larger edit. The sliding window is effective at curbing the negative effects of an edit on MR, because it limits the number of phrases which depend on the edited phrases. For LZ-End, on the other hand, there is no limit on the

number of phrases which can depend on the edited target, other than the number of phrases which follow the edit location.

- **Replacements and deletions at the start of a string have a huge detrimental impact on the LZ-End compression.**
  This is the case even if the replacement or deletion is quite small. This effect rapidly disappears as the edit is made further from the start of the string. The reason for this is that phrases at the start of the string have a huge number of dependents, which occur throughout the parsing.
- **The position of an edit has minimal effect on the MR for LZ-Local.**
  The variance of the MR decreases as the edit is made closer to the end of the string. However, the mean MR has almost no correlation to the position of the edit.
- **Replacements are the sum of insertions and deletions.**
  When an insertion had minimal effect on MR compared to a deletion, the replacement ratio would mirror the deletion. When both insertions and deletions influenced the MR, we would generally see the combination of these effects in the replacement ratio.

7.5 Conclusion and future work

This chapter introduced the LZ-Local parsing and editing and began its evaluation. There are a number of items of future work in order to extend our understanding of this parsing and optimise its efficiency, including:

- Identify or design a data structure which supports `rank` and `select` queries, as well as updates, where the run-time complexity of these functions are not dependent on the size of the structure. This will enable us to define the situations in which the compressed edit for LZ-Local parsing outperforms the raw edit.
- Develop an efficient algorithm for constructing the LZ-Local parsing. In this thesis, we defined the parsing without developing a performant method of constructing it. We have implemented a naïve variant of this parsing; however, developing a performant algorithm remains future work.
- Investigate whether the LZ-Local parsing (or a variant of it) could be proven to be coarsely optimal. If this is possible, can we develop an editing algorithm that maintains the coarse optimality of the parsing?
- Further empirically evaluate the `LZ-Local edit()` function. In this thesis, we only hinted at the case where the inserted string comes from a different source to the compressed string which we are editing. How does inserting a string from a different source to the compressed string affect

the modification ratio?

The next chapter summarises the contributions of this thesis.

Chapter 8

Conclusion

This thesis explored the challenge of editing data in Lempel-Ziv compressed form. This chapter summarises the contributions presented in this thesis, then goes on to discuss future work which can build on the results presented here.

8.1 Summary of contributions

Chapter 1 asked three research questions:

1. Is it possible to use the random-access properties of LZ-End to allow random edits to the compressed data?
2. Is it possible to create an effective LZ compression algorithm that also supports local access?
3. Can one create a locally editable LZ compression?

Chapter 6 answered the first research question by presenting a novel algorithm to edit data in LZ-End-compressed form. The time complexity of the edit is:

$$O((j - i + \bar{l}) \times (\texttt{rank} + \texttt{select}) + (b - a) \times (\Pi.size - b) + \mathcal{F}(S.size)) \quad (6.9)$$

The cost of editing an LZ-End-compressed string is therefore affected by the size of the compressed string ($\Pi.size$), the position of the edit within the string (i, j, a, and b), the length of the longest phrase in the parsing Π, prior to the edit being applied ($\bar{l}$), the size of the edit ($S.size$), as well as the complexity of the `rank`/`select` queries. With this formula not even including the cost of updating the `rank`/`select` data structure, there was huge room for improvement.

Chapter 7 answered the second research question by introducing an LZ-77-style sliding window into the LZ-End parsing. The resulting parsing, referred to as LZ-Local in this thesis, enables local access to the compressed data. This is an improvement on LZ-End parsing, which only allows random access to the compressed data. Furthermore, Chapter 7 went on to answer the final research

question, by presenting an algorithm to locally edit LZ-Local-compressed data. This algorithm has a runtime complexity of

$$O\left(\frac{w \log \bar{m}}{\log w}\right) + \mathcal{F}(pref\_len + 2w + S.size) \tag{7.2}$$

This formula includes the cost of updating the data structure to support `rank` and `select` queries. This means that the cost of a compressed edit is now a function of the window size used in the parsing (w, as well as $pref\_len$, which is strictly less than w), and the size of the edit ($S.size$). The size of the compressed data ($\bar{m}$ = number of LZ-Local phrases prior to or following the edit, whichever is greater) also affects the run-time of the edit; however this is entirely due to the cost of performing the `rank` and `select` queries, as well as updating the supporting data structure. Therefore, an optimised dynamic indexable dictionary will improve this run-time.

Empirical evaluation showed that LZ-Local's compression performance approaches LZ-End's as the window size increases; however, for a given window size, LZ-77 generally compresses strings to a smaller size than LZ-Local.

Further evaluation demonstrated the effect of a compressed edit on the compression ratio. A compressed edit will generally degrade the compression of a string; however, this effect is greatly reduced in LZ-Local compared to LZ-End.

8.2 Final remarks

This thesis represents the author's first attempt at editing LZ-compressed strings; more work remains to be done. Minimal work has been done here on the theoretical evaluation of either the compression performance of the LZ-Local parsing, or of the negative impact an edit can have on compressibility.

Future empirical evaluation of the LZ-Local and LZ-End editing algorithms could include evaluating the effect of edits where the inserted string comes from a different source to the original string. Another line of enquiry could involve empirically analysing the relationship between the compression ratio and the number of dependent phrases each phrase has. This may provide insights into how to improve the editing algorithm's modification ratio.

Future theoretical work relating to the LZ-Local and LZ-End editing algorithms could include:

- Proving (or disproving) the coarse optimality of LZ-Local compression.
- Proving an upper bound on the modification ratio of an edit.
  Could we apply a combinatorial technique like Kempa and Prezza used in [38] to place an upper bound on the number of phrases which could

depend on the target phrases of an edit?

- Generalising the random editing function: LZB and LZ-78 allow random access to the compressed data [53, 49]. Does this mean those parsings allow random edits?

  More generally, investigating applications of Kempa and Prezza's work on string attractors to random editing, in the same way which they applied their work to random access [38].

Finally, future work could investigate improving the LZ-Local parsing. We have already shown how improved dynamic indexable dictionaries supporting fast `rank` and `select` queries can greatly enhance the run-time performance of the LZ-Local editing function.

Further work could be done to improve the LZ-Local parsing: This thesis defined the LZ-Local parsing, but did not present an efficient algorithm to construct it. The LZ-End construction algorithm involves pre-computing data structures such as the BWT, suffix array, *etc.* over the entire string prior to parsing the first phrase. This does not lend itself to efficient parsing with a sliding window. Future work should involve developing a fast algorithm to construct the LZ-Local parsing.

Finally, one could incorporate LZ-SS-style phrases into LZ-Local parsing. Recall from Section 3.3 that LZ-SS parsing is based on LZ-77, but encodes short phrases as a run of innovation symbols [42]. This has two benefits: 1) short phrases are encoded using fewer bits than in LZ-77, and 2) the parsing algorithm can make use of extremely fast hash-tables to identify the best back-reference. The LZ-Local parsing could gain these benefits, in addition to another important benefit: If more symbols are written out explicitly, future phrases have more options to back-reference. Recall that LZ-End and LZ-Local parsings support random access by requiring that each back-reference ends at the innovation-symbol component of a previous phrase. By encoding a short phrase as a run of innovation symbols, future phrases will have more symbols which they could back-reference, thereby improving the compression.

References

[1] A. Lempel and J. Ziv, "On the complexity of finite sequences," *IEEE Trans. Inf. Theory*, vol. 22, no. 1, pp. 75–81, 1976.

[2] S. Kreft and G. Navarro, "LZ77-like compression with fast random access," in *Proceedings of the IEEE Data Compression Conference 24-26 March 2010*, pp. 239–248, IEEE Computer Society, 2010.

[3] R. V. Hartley, "Transmission of information 1," *Bell System technical journal*, vol. 7, no. 3, pp. 535–563, 1928.

[4] C. E. Shannon, "A mathematical theory of communication," *Bell Syst. Tech. J.*, vol. 27, no. 3, pp. 379–423, 1948.

[5] S. R. Kosaraju and G. Manzini, "Compression of low entropy strings with Lempel-Ziv algorithms," *SIAM J. Comput.*, vol. 29, no. 3, pp. 893–911, 1999.

[6] T. M. Cover and J. A. Thomas, *Elements of Information Theory*. Wiley, 2001.

[7] G. K. Wallace, "The JPEG still picture compression standard," *Commun. ACM*, vol. 34, no. 4, pp. 30–44, 1991.

[8] A. N. Kolmogorov, "Three approaches to the quantitative definition of information," *Problems of Information Transmission*, vol. 1, no. 1, pp. 1–7, 1965.

[9] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Commun. ACM*, vol. 30, no. 6, pp. 520–540, 1987.

[10] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[11] J. G. Cleary and I. H. Witten, "Data compression using adaptive coding and partial string matching," *IEEE Trans. Commun.*, vol. 32, no. 4, pp. 396–402, 1984.

[12] J. Duda, K. Tahboub, N. J. Gadgil, and E. J. Delp, "The use of asymmetric numeral systems as an accurate replacement for Huffman coding," in *2015 Picture Coding Symposium (PCS), May 31 - June 3, 2015*, pp. 65–69, IEEE Computer Society, 2015.

[13] C. G. Nevill-Manning and I. H. Witten, "Identifying hierarchical structure in sequences: A linear-time algorithm," *J. Artif. Intell. Res.*, vol. 7, pp. 67–82, 1997.

[14] N. J. Larsson and A. Moffat, "Off-line dictionary-based compression," *Proc. IEEE Computer Society*, vol. 88, no. 11, pp. 1722–1732, 2000.

[15] A. Amir and G. Benson, "Efficient two-dimensional compressed matching," in *Proceedings of the IEEE Data Compression Conference, March 24-27, 1992*, pp. 279–288, IEEE Computer Society, 1992.

[16] M. Farach and M. Thorup, "String matching in lempel-ziv compressed strings," in *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing, 29 May-1 June 1995*, pp. 703–712, ACM, 1995.

[17] L. Gasieniec and W. Rytter, "Almost optimal fully lzw-compressed pattern matching," in *Proceedings of the IEEE Data Compression Conference, March 29-31, 1999*, pp. 316–325, IEEE Computer Society, 1999.

[18] P. Gawrychowski, "Pattern matching in lempel-ziv compressed strings: Fast, simple, and deterministic," in *19th Annual European Symposium on Algorithms, September 5-9, 2011. Proceedings*, vol. 6942 of *Lecture Notes in Computer Science*, pp. 421–432, Springer, 2011.

[19] G. Navarro and M. Raffinot, "A general practical approach to pattern matching over Ziv-Lempel compressed text," in *Combinatorial Pattern Matching, 10th Annual Symposium (CPM), July 22-24, 1999, Proceedings*, vol. 1645 of *Lecture Notes in Computer Science*, pp. 14–36, Springer, 1999.

[20] E. S. de Moura, G. Navarro, N. Ziviani, and R. A. Baeza-Yates, "Direct pattern matching on compressed text," in *String Processing and Information Retrieval: A South American Symposium, SPIRE, September 9-11, 1998*, pp. 90–95, IEEE Computer Society, 1998.

[21] S. Mitarai, M. Hirao, T. Matsumoto, A. Shinohara, M. Takeda, and S. Arikawa, "Compressed pattern matching for SEQUITUR," in *Data Compression Conference, March 27-29, 2001*, p. 469, IEEE Computer Society, 2001.

[22] R. Grossi, R. Raman, S. R. Satti, and R. Venturini, "Dynamic compressed strings with random access," in *Automata, Languages, and Programming - 40th International Colloquium, ICALP, July 8-12, 2013, Proceedings, Part I*, vol. 7965 of *Lecture Notes in Computer Science*, pp. 504–515, Springer, 2013.

[23] K. Tatwawadi, S. S. Bidokhti, and T. Weissman, "On universal compression with constant random access," in *2018 IEEE International Symposium on Information Theory, ISIT 2018*, pp. 891–895, IEEE Computer Society, 2018.

[24] R. Vestergaard, D. E. Lucani, and Q. Zhang, "A randomly accessible lossless compression scheme for time-series data," in *39th IEEE Conference on Computer Communications, (INFOCOM), July 6-9, 2020*, pp. 2145–2154, IEEE Computer Society, 2020.

[25] J. Jansson, K. Sadakane, and W. Sung, "CRAM: compressed random access memory," in *Automata, Languages, and Programming - 39th International Colloquium (ICALP), July 9-13, 2012, Proceedings, Part I*, vol. 7391 of *Lecture Notes in Computer Science*, pp. 510–521, Springer, 2012.

[26] A. Mazumdar, V. Chandar, and G. W. Wornell, "Local recovery in data compression for general sources," in *IEEE International Symposium on Information Theory (ISIT), June 14-19, 2015*, pp. 2984–2988, IEEE Computer Society, 2015.

[27] H. Buhrman, P. B. Miltersen, J. Radhakrishnan, and S. Venkatesh, "Are bitvectors optimal?," *SIAM J. Comput.*, vol. 31, no. 6, pp. 1723–1744, 2002.

[28] P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. R. Satti, and O. Weimann, "Random access to grammar-compressed strings," in *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, January 23-25, 2011*, pp. 373–389, SIAM, 2011.

[29] H. Zhou, D. Wang, and G. W. Wornell, "A simple class of efficient compression schemes supporting local access and editing," in *IEEE International Symposium on Information Theory (ISIT), June 29 - July 4, 2014*, pp. 2489–2493, IEEE Computer Society, 2014.

[30] S. Vatedka and A. Tchamkerten, "Local decoding and update of compressed data," in *IEEE International Symposium on Information Theory (ISIT), July 7-12, 2019*, pp. 572–576, IEEE Computer Society, 2019.

[31] S. Vatedka and A. Tchamkerten, "Local decode and update for big data compression," *IEEE Trans. Inf. Theory*, vol. 66, no. 9, pp. 5790–5805, 2020.

[32] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *41st Annual Symposium on Foundations of Computer Science (FOCS), 12-14 November 2000*, pp. 390–398, IEEE Computer Society, 2000.

[33] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi, "LZ77-based self-indexing with faster pattern matching," in *Theoretical Informatics - 11th Latin American Symposium (LATIN), March 31 - April 4, 2014. Proceedings*, vol. 8392 of *Lecture Notes in Computer Science*, pp. 731–742, Springer, 2014.

[34] W. Hon, T. W. Lam, K. Sadakane, W. Sung, and S. Yiu, "Compressed index for dynamic text," in *Proceedings of the IEEE Data Compression Conference 23-25 March 2004*, pp. 102–111, IEEE Computer Society, 2004.

[35] S. Kreft and G. Navarro, "Self-indexing based on LZ77," in *Combinatorial Pattern Matching - 22nd Annual Symposium (CPM), June 27-29, 2011. Proceedings*, vol. 6661 of *Lecture Notes in Computer Science*, pp. 41–54, Springer, 2011.

[36] G. Navarro and N. Prezza, "Universal compressed text indexing," *Theor. Comput. Sci.*, vol. 762, pp. 41–50, 2019.

[37] T. Nishimoto, T. I, S. Inenaga, H. Bannai, and M. Takeda, "Dynamic index and LZ factorization in compressed space," *Discret. Appl. Math.*, vol. 274, pp. 116–129, 2020.

[38] D. Kempa and N. Prezza, "At the roots of dictionary compression: string attractors," in *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, June 25-29, 2018*, pp. 827–840, ACM, 2018.

[39] D. Dubé and V. Beaudoin, "Improving LZ77 bit recycling using all matches," in *IEEE International Symposium on Information Theory (ISIT), July 6-11, 2008*, pp. 985–989, IEEE Computer Society, 2008.

[40] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inf. Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[41] A. D. Wyner and J. Ziv, "The sliding-window Lempel-Ziv algorithm is asymptotically optimal," *Proc. IEEE Computer Society*, vol. 82, no. 6, pp. 872–877, 1994.

[42] J. A. Storer and T. G. Szymanski, "Data compression via textual substitution," *J. ACM*, vol. 29, no. 4, pp. 928–951, 1982.

[43] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Trans. Inf. Theory*, vol. 24, no. 5, pp. 530–536, 1978.

[44] K. Sadakane and H. Imai, "Improving the speed of LZ77 compression by hashing and suffix sorting," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 83, no. 12, pp. 2689–2698, 2000.

[45] J. Kärkkäinen, D. Kempa, and S. J. Puglisi, "Lightweight lempel-ziv parsing," in *Experimental Algorithms, 12th International Symposium (SEA), June 5-7, 2013. Proceedings*, vol. 7933 of *Lecture Notes in Computer Science*, pp. 139–150, Springer, 2013.

[46] M. Crochemore and L. Ilie, "Computing longest previous factor in linear time and applications," *Inf. Process. Lett.*, vol. 106, no. 2, pp. 75–80, 2008.

[47] D. Belazzougui and S. J. Puglisi, "Range predecessor and Lempel-Ziv parsing," in *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, January 10-12, 2016*, pp. 2053–2071, SIAM, 2016.

[48] T. A. Welch, "A technique for high-performance data compression," *Computer*, vol. 17, no. 6, pp. 8–19, 1984.

[49] A. Dutta, R. Levi, D. Ron, and R. Rubinfeld, "A simple online competitive adaptation of Lempel-Ziv compression with efficient random access support," in *Proceedings of the IEEE Data Compression Conference, March 20-22, 2013*, pp. 113–122, IEEE Computer Society, 2013.

[50] A. Amir, G. Benson, and M. Farach, "Let sleeping files lie: Pattern matching in z-compressed files," *J. Comput. Syst. Sci.*, vol. 52, no. 2, pp. 299–307, 1996.

[51] T. Tao and A. Mukherjee, "LZW based compressed pattern matching," in *Proceedings of the IEEE Data Compression Conference 23-25 March 2004*, p. 568, IEEE Computer Society, 2004.

[52] G. Navarro and J. Tarhio, "Boyer-Moore string matching over Ziv-Lempel compressed text," in *Combinatorial Pattern Matching, 11th Annual Symposium (CPM), June 21-23, 2000, Proceedings*, vol. 1848 of *Lecture Notes in Computer Science*, pp. 166–180, Springer, 2000.

[53] M. Banikazemi, "LZB: data compression with bounded references," in *Proceedings of the IEEE Data Compression Conference 16-18 March 2009*, p. 436, IEEE Computer Society, 2009.

[54] T. Bell, *A unifying theory and improvements for existing approaches to text compression*. PhD thesis, University of Canterbury, 1986.

[55] D. Kempa and D. Kosolobov, "LZ-End parsing in linear time," in *25th Annual European Symposium on Algorithms (ESA), September 4-6, 2017*, vol. 87 of *LIPIcs*, pp. 53:1–53:14, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[56] D. Kempa and D. Kosolobov, "LZ-End parsing in compressed space," in *Proceedings of the IEEE Data Compression Conference, April 4-7, 2017*, pp. 350–359, IEEE Computer Society, 2017.

[57] K. Sadakane and R. Grossi, "Squeezing succinct data structures into entropy bounds," in *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 22-26, 2006*, pp. 1230–1239, ACM Press, 2006.

[58] R. González and G. Navarro, "Statistical encoding of succinct data structures," in *Combinatorial Pattern Matching, 17th Annual Symposium, CPM 2006, Barcelona, Spain, July 5-7, 2006, Proceedings*, vol. 4009 of *Lecture Notes in Computer Science*, pp. 294–305, Springer, 2006.

[59] P. Ferragina and R. Venturini, "A simple storage scheme for strings achieving entropy bounds," *Theor. Comput. Sci.*, vol. 372, no. 1, pp. 115–121, 2007.

[60] P. Ferragina and G. Manzini, "Indexing compressed text," *J. ACM*, vol. 52, no. 4, pp. 552–581, 2005.

[61] R. Raman, V. Raman, and S. S. Rao, "Succinct indexable dictionaries with applications to encoding k-ary trees and multisets," in *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002*, pp. 233–242, ACM/SIAM, 2002.

[62] G. Nong, S. Zhang, and W. H. Chan, "Linear suffix array construction by almost pure induced-sorting," in *Proceedings of the IEEE Data Compression Conference, 16-18 March 2009*, pp. 193–202, IEEE Computer Society, 2009.

[63] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches," in *Proceedings of the First Annual ACM-SIAM Symposium*

on Discrete Algorithms, pp. 319–327, Society for Industrial and Applied Mathematics, 1990.

[64] D. Gusfield, *Algorithms on Strings, Trees, and Sequences – Computer Science and Computational Biology*. Cambridge University Press, 1997.

[65] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, "Replacing suffix trees with enhanced suffix arrays," *J. Discrete Algorithms*, vol. 2, no. 1, pp. 53–86, 2004.

[66] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," in *Digital SRC Research Report*, Citeseer, 1994.

[67] G. Manzini, "An analysis of the Burrows-Wheeler transform," *J. ACM*, vol. 48, no. 3, pp. 407–430, 2001.

[68] J. Fischer and V. Heun, "A new succinct representation of rmq-information and improvements in the enhanced suffix array," in *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies, First International Symposium (ESCAPE), April 7-9, 2007, Revised Selected Papers*, vol. 4614 of *Lecture Notes in Computer Science*, pp. 459–470, Springer, 2007.

[69] T. C. Bell, J. G. Cleary, and I. H. Witten, *Text compression*. Prentice-Hall, Inc., 1990.

[70] R. Arnold and T. C. Bell, "A corpus for the evaluation of lossless compression algorithms," in *Proceedings of the IEEE Data Compression Conference (DCC), March 25-27, 1997*, pp. 201–210, IEEE Computer Society, 1997.

[71] P. Ferragina and G. Navarro, "The Pizza & Chili Corpus," 2007.

[72] W. Ebeling, R. Steuer, and M. Titchener, "Partition-based entropies of deterministic and stochastic maps," *Stochastics and Dynamics*, vol. 1, no. 01, pp. 45–61, 2001.

[73] M. R. Titchener, P. M. Fenwick, and M. C. Chen, "Towards a calibrated corpus for compression testing," in *Proceedings of the IEEE Data Compression Conference (DCC), March 29-31, 1999*, p. 554, IEEE Computer Society, 1999.

[74] U. Speidel, M. Titchener, and J. Yang, "How well do practical information measures estimate the Shannon entropy?," in *Fifth International Symposium on Communication Systems, Networks, and Digital Signal Processing (CSNDSP)*, 2006.

[75] J. P. Crutchfield and N. H. Packard, "Symbolic dynamics of noisy chaos," *Physica D: Nonlinear Phenomena*, vol. 7, no. 1-3, pp. 201–223, 1983.

[76] D. Roodt, U. Speidel, V. Kumar, and R. K. Ko, "On random editing in LZ-End," in *Proceedings of the IEEE Data Compression Conference (DCC), March 23-26, 2021*, pp. 366–366, IEEE Computer Society, 2021.

[77] P. Ferragina and G. Vinciguerra, "The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds," *Proc. VLDB Endow.*, vol. 13, no. 8, pp. 1162–1175, 2020.

[78] R. González and G. Navarro, "Improved dynamic rank-select entropy-bound structures," in *Theoretical Informatics, 8th Latin American Symposium (LATIN), April 7-11, 2008, Proceedings*, vol. 4957 of *Lecture Notes in Computer Science*, pp. 374–386, Springer, 2008.

[79] M. Patrascu and M. Thorup, "Dynamic integer sets with optimal rank, select, and predecessor search," in *55th IEEE Annual Symposium on Foundations of Computer Science, October 18-21, 2014*, pp. 166–175, IEEE Computer Society, 2014.

[80] S. Dönges, S. J. Puglisi, and R. Raman, "On dynamic bitvector implementations," in *Proceedings of the IEEE Data Compression Conference, March 22-25, 2022*, pp. 252–261, IEEE Computer Society, 2022.

Appendix: A note on alphabets

Recall the strange observations in the experimental results when making large insertions into low, medium and high entropy strings (see Figures 7.39 to 7.41, repeated in Figure A.1): Large insertions into high entropy strings consistently result in large modification ratios. Large insertions into medium entropy strings, however, sometimes result in large MRs, but sometimes result in MRs near 1. Larger insertions into low entropy strings have no effect on the MR, except for two outlying strings, which seem to have the same correlation between insertion size and MR as high entropy strings. These observations apply to strings compressed by LZ-End, and are not present in the strings compressed using LZ-Local.

The results for medium-entropy insertions (Figure A.1b) were particularly alarming, because all measurements were on either end of the box-and-whisker plot. That is, all MRs were either very close to 1.0, or very close to 1.025. There were no values in between.

These results are an artefact of a different alphabet being used to create the strings of calibrated entropy compared to the alphabet used by the implementation of the LZ compressors. Recall that calibrated entropy strings were generated by a binary partition; this means that the string consists of symbols from the alphabet $\mathbb{A} = \{0, 1\}$. The LZ compressor, on the other hand, interpreted this bit-stream as a string of bytes, with an alphabet of size 256: $\mathbb{A}' = \{\texttt{0x00, 0x01, 0x02, ..., 0xFE, 0xFF}\}$.

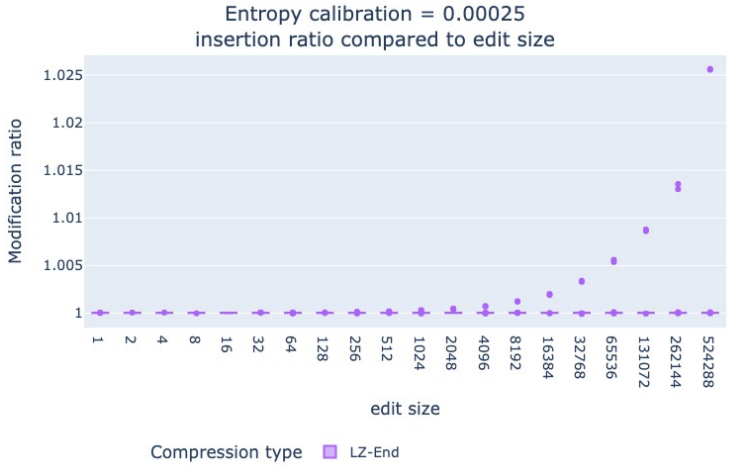
This is an important difference: consider a source which produces a bit-stream of alternating 0's and 1's:

$$B = \texttt{0101010101}\ldots$$

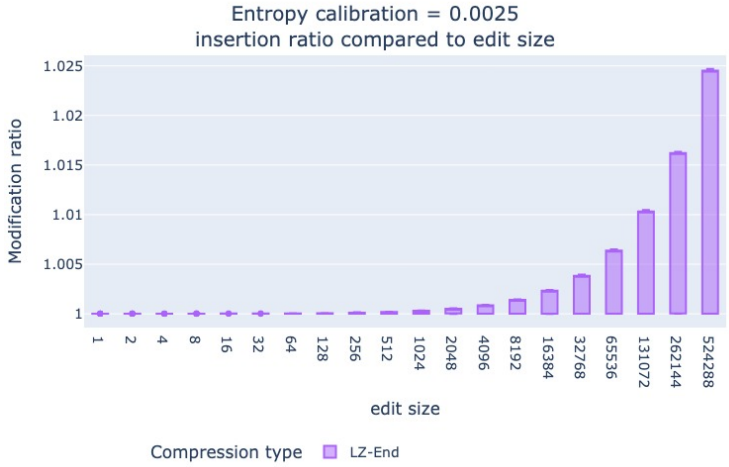
Imagine that we wish to read this as a sequence of bytes. If we do that, we get

$$B' = \texttt{0x55 0x55 0x55}\ldots$$

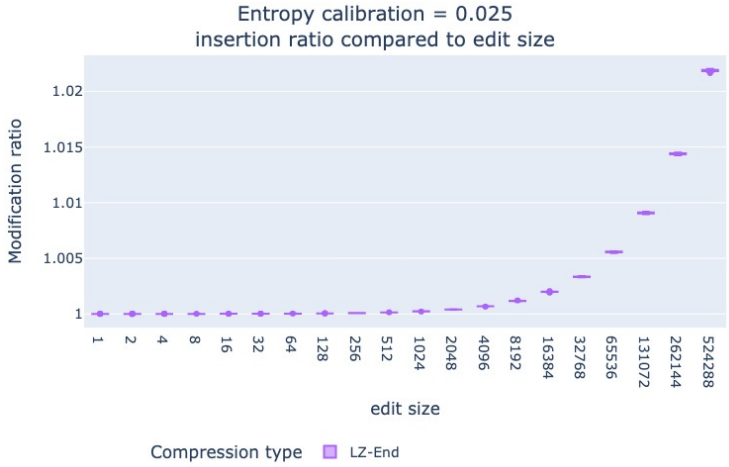
If, however, we read from the same source, but with a 1 in the first position,



(a) Insertion: low entropy



(b) Insertion: medium entropy



(c) Insertion: high entropy

Figure A.1

we get a very different byte stream:

$$B'' = \texttt{0xAA 0xAA 0xAA}\ldots$$

Although byte streams B' and B'' came from the same source, a LZ parser will not use any phrases from B' to compress the bytes in B''.

The low entropy strings consist of an uneven distribution of bytes, while the high entropy strings have a closer to uniform distribution of bytes. This means that two byte strings drawn from a high-entropy source of bits will have many symbols in common. Conversely, two byte strings drawn from the same low-entropy source of bits will have a low chance that the most frequently-occurring byte in each string is the same.

Consider two byte strings S and S', taken from the same binary source. They will by definition have the same distribution of symbols. However, the "labelling" of these distributions may not be the same: That is, the most commonly-occurring symbol in S will occur with the same frequency as the most commonly-occurring symbol in S'. However, these two symbols may not be the same. This is an artefact of the conversion of a binary stream to a byte string.

There are therefore two possible scenarios:

1. The commonly-occurring symbols of S and S' have a high overlap. That is, a symbol that occurs frequently in S also appears frequently in S', and one that occurs infrequently in S also appears infrequently in S'.
2. The commonly-occurring symbols of S and S' have little overlap. That is, commonly-occurring symbols in S infrequently appear in S' and vice versa.

We wish to insert the string S' into S, and make a comparison between performing a compressed edit and a raw edit.

In the first scenario, where the symbols of S and S' have high overlap, the raw edit will be result in a more effective compression than the compressed edit. This is because the raw edit will allow the part of S following the insertion location to be better compressed due to the insertion. On the other hand, the compressed edit will mean that the compression of S does not benefit from the compression of S'. This means that the modification ratio $\mu = \frac{\Pi'_e.size}{\Pi'.size}$ will be larger than 1. Recall that Π'_e is the parsing of the compressed edit, while Π' is the parsing of the raw edit.

In the second scenario, where the symbols of S and S' have little overlap, the string S' does not provide many phrases to aid in the LZ-compression of S. Therefore, applying the raw edit, and compressing the result, does not benefit us in any way: Although the LZ phrases provided by the compression of S' are

available for use by the remaining part of S, these phrases are not relevant, and go largely unused. In this case, we expect the modification ratio to be close to 1.

A high entropy source of S and S' will result in a reasonably flat distribution of symbols. Therefore, the first case will apply. A low entropy source of S and S' will have a very uneven distribution of symbols. There is a small chance (depending on the size of the alphabet and the shape of the distributions) that these overlap. If they do overlap, then the first case will apply. However, if the distributions do not overlap, the second case will apply.

This is what we are seeing in our experiments (Figures 7.39 to 7.41): In the high-entropy case, we see the high overlap of symbol distributions for all strings. In the low-entropy case, we get no overlap in distributions for most strings. However, we get two (out of the 30 strings) with highly overlapping distributions. These two strings show a correlation between insertion size and MR that is similar to the low entropy case. In the medium entropy case, we get mixed results, whereby more files resemble the high-entropy relationship between insertion size and MR than for the low-entropy case.

To prove this point, let us consider the low-entropy strings. Each of these 30 strings has only 3 bytes which have more than one occurrence in the string. In the case of the first low-entropy string, the only bytes that occur more than once are `0x5D, 0xD5,` and `0xDD`. However, the string that we insert has a different set of three bytes: `0xAD, 0xAE,` and `0xEA`. The two low-entropy strings which result in the extremely outlying MRs are the only two strings for which the three bytes are `0xAD, 0xAE,` and `0xEA`.

Correct interpretation of the experiments

This now raises the question: How should we interpret the experiments?

The high-entropy case (Figure A.1c) can be taken at face value. The size of the insertion is strongly correlated to the size of the MR.

The medium and low-entropy strings are less useful. We do, however, gain an interesting observation we had not bargained for: the case where the inserted string does not come from the same source as the existing string. This may happen, for example, when inserting an image into a compressed archive which previously only contained text files. In this case, we see that the MR is likely to remain close to 1.

On the other hand, we would expect that for LZ-End, the size of the insertion into a string of any entropy will determine the modification ratio. However, we cannot currently quantify this relationship, as the two data-points we have in the low-entropy case are not enough to draw any conclusions.

Why LZ-Local did not have such outliers

Another question that may be raised is: Why did LZ-Local not have such outliers? There are two reasons for this:

1. The size of the edit dwarfed the size of the window, so that, regardless of whether S and S' come from the same source, the LZ phrases of S' will only affect the compression of a small portion of S. The largest window size we used was 65,536 bytes, while the largest edit we made was 524,288 bytes. Therefore, any potential gain in compressing S as a result of inserting S' was very minor; hence the consistent results across all entropy levels for insertions into LZ-Local-compressed strings.
2. LZ-Local edit algorithm uses the compressed form of S to aid in compressing the inserted string S'.

On the use of calibrated entropy strings

This highlights a limitation of using a binary partition to the logistic map. Future work should include developing a source of calibrated entropy strings that generate the same alphabet used by the LZ parser (or any other compression algorithm). This could be resolved by adapting the implementations of the LZ compressors that we use: if the compressor reads as a binary alphabet, this problem would easily be resolved. However, this would add significant overheads to the runtime and memory of the experiments, which would make the current set of experiments infeasible.

Conclusion and learnings

The key lessons from this investigation are:

- One must be careful when the source of one's string uses a different alphabet to that used by the LZ parser. In our experiments, the sources of our strings used a binary alphabet, whereas the LZ parsers interpreted these strings as bytes.
- The MR for insertions into LZ-End-compressed strings is dependent on the size of the insertion, if the inserted string comes from the same source as the string we are inserting into. We cannot quantify this exact relationship for low and medium entropy strings though.
- The MR for insertions into LZ-End-compressed strings, when the inserted string comes from a different source to the string we are inserting into is likely to be very close to 1. This is regardless of the size of the inserted string.

Index