Repetition-Aware Lossless Compression

（反復構造のための可逆圧縮）

Isamu Furuya

August 2020

Division of Computer Science and Information Technology

Graduate School of Information Science and Technology

Hokkaido University

Abstract

This thesis studies lossless compression techniques for repetitive data. Lossless compression is a type of data compression that allows restoring the original information completely from compressed data. Today's ever-growing information technology industries involve the enormous data growth, and then an efficient method managing large data is desired. Whereas, these large data in our society are in many cases highly repetitive, that is, most of their fragment parts can be obtained from others occurring in other positions in the data with a few modifications. Managing large repetitive data efficiently is getting attention in many fields and demands for a good compression method for such repetitive data are increasing. A repetition-aware compression technique allows to manage these large data more efficiently and this study contributes to the technique. The term repetition-aware means high effectiveness for repetitiveness. Our approaches to repetition-aware compression are through the grammar compression scheme that constructs a formal grammar that generates a language consisting only of the input data. Grammar compression have been preferable over other lossless compression techniques because of some profitable properties including practical high compression performance for repetitive data. The heart of this study is to develop a grammar compression method that aims to construct a small sized formal grammar from the input data.

We discuss on three grammar compression frameworks whose differences are the formal grammars used as the description of the compressed data. We consider a context-free grammar (CFG), a run-length context-free grammar (RLCFG), and a functional program described by λ-term in Chapter 3, 4, and 5, respectively.

In Chapter 3, we approach to the problem of repetition-aware compression on CFG-based grammar compression. We analyze a famous algorithm, RePair, and on the basis of the analysis, we design a novel variant of RePair, called MR-RePair. We implement MR-RePair and experimentally confirm the effectiveness of MR-RePair especially for highly repetitive texts.

In Chapter 4, we address further improvement of compression performance via the framework of RLCFG-based grammar compression. In the chapter, we design a compression algorithm using RLCFG, called RL-MR-RePair. Furthermore, we propose an encoding scheme for MR-RePair and RL-MR-RePair. The experimental results demonstrate the high compression performance of RL-MR-RePair and the proposed encoding scheme.

In Chapter 5, we study on the framework of higher-order compression, which is a grammar compression using a λ-term as the formal grammar. We present a method to obtain a compact λ-term representing a natural number. Obtaining a compact representation of natural numbers can improve the compression effectiveness of repetition, the most fundamental repetitive structure. For given natural number n, we prove that the size of the obtained λ-term becomes $\mathcal{O}(\mathrm{slog}_2 n)$ in the best case and $\mathcal{O}(\mathrm{slog}_2 n)^{\log n/\log\log n}$ in the worst case.

Acknowledgements

The completion of this work is due to the support of many people.

Firstly, I would like to express my sincere gratitude to my supervisor, Hiroki Arimura. He has advised me a lot of things not only about how to advance research, but also about how to behave as a researcher.

I would also like to express thanks to my former supervisor, Takuya Kida, who is currently a professor at Hokkai-Gakuen University. He has ensured my research activities to be conducted properly from the beginning of my Bachelor's degree program. His continued support has made much of this work possible.

I am deeply grateful to the past and present members of Information Knowledge Network laboratory at Hokkaido University. They have supported me in a lot of different ways. Takuya Takagi helped me and gave me a lot of good advice when he was in the laboratory, and even after he graduated. We discussed many ideas, and they have been very helpful in my research work. Yu Manabe, the secretary of the laboratory, has supported my laboratory life in many ways, including arrangements for trips.

Some results in this thesis are the products of collaboration with Yuto Nakashima and Shunsuke Inenaga at Kyushu University and Hideo Bannai at Tokyo Medical and Dental University. I am fortunate to have had opportunities to work with them. They have always kindly supported me during writing our paper, which is the previous

version of the work. Their proper comments certainly made our work better.

Finally, I would like to thank my parents and brother for their support and caring.

Contents

1 Introduction **9**

1.1 Background 9

1.2 Research Goals 13

1.3 Contributions 14

1.4 Related Studies 16

1.5 Organization 17

2 Preliminaries **19**

2.1 Texts 19

2.1.1 Basic Notations and Terms on Texts 19

2.1.2 Maximal Repeats 20

2.1.3 Repetitions and Runs 20

2.1.4 Bit Encoding Methods for Texts 20

2.2 Grammars 21

2.2.1 Context-Free Grammars (CFGs) 21

2.2.2 Parse-Trees 22

2.2.3 Run-Length Context-Free Grammars (RLCFGs) 22

2.3 Grammar Compression 22

2.3.1 Compression using a CFG 22
2.3.2 Compression using a RLCFG 23
2.3.3 RePair algorithm 23
2.4 Model of Computation 24
2.4.1 Word RAM 24

3 Grammar Compression based on Maximal Repeats 27
3.1 Introduction 28
3.1.1 Contributions 29
3.1.2 Organization 30
3.2 Analysis of RePair 30
3.2.1 RePair and Maximal Repeats 30
3.2.2 MR-Order 33
3.2.3 Greatest Size Difference of RePair 36
3.3 Proposed Method 38
3.3.1 Naïve-MR-RePair 38
3.3.2 MR-RePair 45
3.4 Experiments 50
3.5 Conclusions 53

4 Grammar Compression with Run-Length Rules 57
4.1 Introduction 58
4.1.1 Contributions 59
4.1.2 Organization 60
4.2 Proposed Method 60
4.2.1 Algorithm 60

4.2.2 Implementation 61
4.3 Bit Encoding 63
4.3.1 A Previous Effective Method for RePair 63
4.3.2 Encoding via Post-Order Partial Parse Tree (POPPT) 65
4.3.3 Combination of POPPT and PGE 66
4.4 Experiments 66
4.4.1 Grammar Construction 66
4.4.2 Encoding the Grammars 67
4.5 Conclusions 69

5 Compaction of Natural Numbers for Higher-Order Compression 77
5.1 Introduction 78
5.1.1 Contributions 79
5.1.2 Organization 80
5.2 Preliminaries 80
5.2.1 Tetration and Super-Logarithm 81
5.2.2 Lambda Terms 81
5.2.3 Church Numerals 83
5.2.4 Binary Expression of Natural Numbers on λ-Terms 84
5.3 Proposed Method 84
5.3.1 Basic Idea 84
5.3.2 Tetrational Arithmetic Expression (TAE) 85
5.3.3 Translation from TAE to λ-Term 86
5.3.4 Recursive Tetrational Partitioning (RTP) 88
5.3.5 Further Compaction 94
5.4 Application to Higher-Order Compression and Comparative Experiments 99

5.5 Conclusions 101

6 Conclusions **105**

6.1 Summary 105

6.2 Towards the Future 106

Chapter 1

Introduction

1.1 Background

Data compression is the technique for representing the redundant information in data in a more economical way. Much of information is ordinarily embodied as a data in its raw form and this raw material tends to be large because it holds much redundancy in many cases. Today's ever-growing information technology industries in our society involve the enormous data growth, and then an efficient method managing such data is desired. Data compression is one of the enabling technologies for the requirement. Data compression reduces the storing and transmitting costs of the vast quantities of data and allows us to use such data more efficiently.

Extracting the essential information from an input data is a basic principle of data compression. The origin of challenges for the question of what is the essential information of a datum goes back to information entropy (also known as Shannon entropy) [88, 89] in 1948 or Kolmogorov complexity [54, 55, 91, 92, 23] in the 1960s. These two concepts provide quantitative measures of information based on different

ideas. In information entropy, the quantity of information associated with an event A, which is a set of outcomes of some random experiment, is defined as

$$\log \frac{1}{P(A)} = -\log P(A),$$

where $P(A)$ is the probability that A will occur. This means that if the probability of an event is low, the amount of information contained in the event is high, and vice versa. Kolmogorov complexity gives another measure. Let s be a sequence representing a datum. Then, in Kolmogorov complexity, the quantity of information associated with s is defined as the size of the program which generates s. Note that we do not have to specify the programming language because of the invariance theorem such that a program in one language can be translated to that in another language at fixed cost. These two approaches are now regarded as the roots of data compression. The underlying philosophy of data compression is to represent data in the size aligned with the amount of its essential information.

Lossless compression is a type of methods for data compression which allows restoring the original information completely from compressed data. In data compression, two kinds of techniques have been developed; one involves some loss of information, and the other involves no loss of information. The former is called lossy compression, which aims for high compression performance by getting rid of some information in the original data. Since some information is lost, generally, the exact original data cannot be recovered from compressed data in lossy compression. Lossless compression is the latter. Nowadays, the targets of lossless compression include many kinds of data such as revision histories of wiki pages, server access logs, point of sale data (POS), genome databases, pathology images, satellite maps, and so on.

Dictionary-based compression (or *dictionary compression*) is a technique of lossless compression that uses a dictionary for compression. A dictionary consists of a set of

patterns and a codeword for each pattern. If a pattern registered in the dictionary is appeared in data, the pattern is represented by the corresponding codeword. Till now, this technique has been the most successful solution to the problem of lossless compression. Dictionary-based compression originated in the papers by Ziv and Lempel in 1977 [105] and 1978 [106], which present famous algorithms known as LZ77 and LZ78, respectively. More precisely, this two is the origin of one class of dictionary-based compression, called adaptive (or dynamic) dictionary compression. Ahead of them, the basic idea of dictionary-based compression itself was presented [85] and this method is classified in a type called static dictionary compression. As the name suggests, static dictionary compression uses a static dictionary which is generally prepared in advance. In contrast, adaptive dictionary compression constructs a dictionary adaptively to the input data.

In dictionary-based compression, regardless of static or adaptive, the frequencies of the patterns registered in the dictionary affect the compression performance; that is, if the patterns occur frequently in the input data, dictionary compression will provide small sized compressed data. In fact, dictionary-based compression reduces redundancy by focusing on repetitiveness in the data. We say that a datum is repetitive if an element itself or similar one occur in the datum repeatedly. The large data used in the society are in many cases highly repetitive, that is, most of their fragment parts can be obtained from others occurring in other positions in the data with a few modifications. A representative example is a genome database as used in the 1000 Genome Project [32], namely, a genome sequence collection storing many genomes from the same species. Przeworski et al. [80] reported that genome sequences of humans differ by 0.1%. The important point is that repetitive structures have much redundancies. There is a potential for drastically compressive for these data. The term *repetition-*

aware means high effectiveness for repetitiveness. Dictionary-based compression is now a very reasonable approach to repetition-aware lossless compression.

Grammar compression is a method for dictionary-based compression which uses a formal grammar as the dictionary model. The dictionary is generally constructed adaptively, in more detail, grammar compression converts the input data into a formal grammar that generates a language consisting only of the data. After LZ77 and LZ78 in 1977 and 1978, several algorithms for dictionary compression had been presented [10, 52, 50, 73, 74] and through the studies, beneficial properties of formal grammars as the dictionary model had been revealed. Then, grammar compression was established as a compression scheme by Kieffer and Yang in 2000 [51]. While there are some other successful approaches to lossless compression such as LZ77 and Burrows-Wheeler Transform (BWT) [22], grammar compression have been preferable over these techniques because of its implementation efficiency and practical high compression performance especially for repetitive data. Also, the manageability of compressed data is another important advantage of grammar compression. That is, grammar compression allows accessing information of the original data directly in compressed form [21]. This is suitable to be used as the compressor building a compressed data structure that represents original data in small space while simulating direct access to its information. Now the compressed data structures are getting attention in various fields underlying the motivation of managing large data in compressed form with execution time and memory space depending on its compressed size. Under these circumstances, there are demands for more efficient grammar compression algorithms.

In grammar compression, a context-free grammar (CFG) is mainly used as the formal grammar, namely, as the dictionary model. Meanwhile, techniques using other types of grammars for the model have been also considered [63, 49]. In lossless com-

pression, better modeling for an input data leads better compression performance. Development of an effective compression method for the inherent structure of the input data is a profitable approach. Theoretically, we can optimize the average codeword length for the input data by universal source coding [30], even if we do not know the exact model for the input beforehand. However, indentifying the inherent structure of the input and using an appropriate model for the structure realize much more efficient compression in practical uses attributing the finite sizes of computer memories and input data. In 2012, Kobayashi et al. [53] proposed a compression scheme called *higher-orden compression*, which use a functional program described by a λ-term as the dictionary model. Functional programs are Turing complete and they are classified in Type-0 grammars in Chomsky hierarchy [25], while CFGs are categorized in Type-2 grammars. This means that a smaller description of compressed data model can be exist in functional programs than in CFGs. At the same time, this implies that the search for a small description is more difficult in functional programs because of the larger solution space compared to that of CFGs. Currently, higher-order compression is one of the most seminal developing techniques in the research field of lossless compression; the futuristic high compression performance is expected but various algorithmic challenges are remained.

1.2 Research Goals

To develop a well performing compression method is one of the major research goals in lossless compression. For estimating the performance of a lossless compression technique, there are some indicators including compression ratio, execution time, memory usage, online algorithm, and manageability of the compressed data. This thesis aims

to improve the compression ratio of lossless compression for repetitive data. As stated in Section 1.1, the large data currently used in our society are in many cases highly repetitive and demands for a good compression method for such repetitive data are increasing day by day. A repetition-aware compression technique allows to manage these large data more efficiently and this study contributes to the technique.

1.3 Contributions

Our approaches to repetition-aware compression are through the grammar compression scheme. The heart of our study is to develop a compression method that aims to construct a small sized formal grammar from the input data.

In Chapter 3, we discuss the compression methods for repetitive data on the framework of CFG-based grammar compression. This chapter presents an analysis of RePair [56], a grammar compression algorithm known for its high compression performance. We show that the main process of RePair, that is, the step by step substitution of the most frequent symbol pairs, works within the corresponding most frequent maximal repeats. We reveal the relation between maximal repeats and grammars constructed by RePair and we introduce a concept about selection orders of maximal repeats, called MR-order, which enables to explain an unrevealed behavior of RePair algorithm. Furthermore, we define the problem of determining the greatest size difference between possible outcomes of RePair as GSDRP and we establish a lower bound of the GSDRP. Moreover, on the basis of above analyses, we further propose a novel variant of RePair, called MR-RePair, which considers the one-time substitution of the most frequent maximal repeats instead of the consecutive substitution of the most frequent pairs. The results of the experiments comparing the size of constructed gram-

mars and execution time of RePair and MR-RePair on several text corpora demonstrate that MR-RePair constructs more compact grammars than RePair does, especially for highly repetitive texts.

In Chapter 4, we address further improvement of compression performance via the framework of run-length CFG-based grammar compression. Run-length CFG (RLCFG) [45, 76] is an extension of CFG. The properties of RLCFG have been studied [18, 36] and its usefulness for grammar compression as the dictionary model have been revealed. Theoretically, RLCFG improves the effectiveness of CFG-based grammar compression, but whether this is realized in practice remains undiscovered because, to our knowledge, no compression algorithms for the RLCFG scheme have been developed. Here, we design a compression algorithm using RLCFG, called RL-MR-RePair, which follows RePair and MR-RePair for the CFG scheme. Experimentally, we show that RL-MR-RePair constructs smaller grammars for repetitive datasets than either RePair or MR-RePair. Furthermore, we propose an encoding scheme for grammars. The scheme was originally designed for RL-MR-RePair but is directly applicable to MR-RePair. In comparative experiments from grammar construction to final bit encoding, we evaluate the performance of RePair, MR-RePair, and RL-MR-RePair. The experiments confirm the high compression performance of RL-MR-RePair with the proposed encoding method on real repetitive datasets.

In Chapter 5, we present a compression algorithm for repetitions on the framework of higher-order compression. A repetition is the most fundamental repetitive structure. Obtaining a compact representation of natural numbers can improve the compression effectiveness of repetition because natural number is one of the essential elements of repetition. Higher-order compression uses a λ-term as its dictionary model and Church numerals are unary representations of natural numbers on the scheme of

λ-terms. Here, we address the problem of compaction of Church numerals. We propose a novel decomposition scheme from a given natural number into an arithmetic expression using tetration, which enables us to obtain a compact representation of lambda terms that leads to the Church numeral of the natural number. For given natural number n, we prove that the size of the lambda term obtained by the proposed method is $\mathcal{O}((\text{slog}_2 n)^{(\log n/\log\log n)})$. Moreover, we experimentally confirmed that the proposed method outperforms binary representation of Church numerals on average, when n is less than approximately 10,000.

1.4 Related Studies

Data compression is a fundamental research area of theoretical information science. For lossless compression and related problems, many different approaches have been proposed till today. Fortunately, several excellent books covering these approaches have been published [93, 15, 72, 86, 103, 84, 65, 69].

Managing large repetitive data efficiently is getting attention in many fields and several approaches have been proposed [59, 90, 60, 68, 36]. From the viewpoint of lossless compression, some remarkable results discovering the theoretical effectiveness of dictionary-based compression on repetitive data have been presented [48, 36].

In grammar compression, a smaller set of rules gives a better compression performance. Finding the optimal grammar is an NP-hard problem [24] but several heuristic algorithms have been proposed for its solution [56, 62, 64, 75, 78, 96, 38, 35]. Obtaining a smaller grammar is of great importance regarding to the compressed data structures. For several problems such as computing edit distance, subsequence matching, or q-gram mining, a lot of algorithms that work on data in compressed form depending on the com-

pressed sizes have been proposed in the past decade [27, 28, 39, 41, 16, 43, 46, 17, 57, 82].

For higher-order compression, Yaguchi et al. [104] proposed an efficient algorithm. They state in [104] that their method often achieves better performance than a grammar compression with regard to compression performance. For bit encoding of λ-terms, Tromp [100] proposed a method for untyped λ-terms. Also, Takeda et al. [99] presented a efficient encoding scheme for simply-typed λ-terms.

1.5 Organization

The remainder of this thesis is organized as follows. In Chapter 2, we provide some notations and definitions to be used through this thesis. In Chapter 3, we analyze RePair algorithm and present MR-RePair algorithm. Following this, in Chapter 4, we propose another practical grammar compression algorithm, called RL-MR-RePair. In Chapter 5, we address the problem of compaction of Church numerals for higher-order Compression. Finally, the thesis concludes with Chapter 6, which also suggests future directions for this work.

Chapter 2

Preliminaries

In this chapter, we provide basic notations and definitions to be used in the following chapters. The notations and definitions not found here will be found in appropriate textbooks in algorithms [9, 29, 31, 47], information theory [30], and data compression [93, 15, 72, 86, 103, 84, 65, 69].

2.1 Texts

2.1.1 Basic Notations and Terms on Texts

Let Σ be an *alphabet*, that is, an ordered finite set of symbols. An element $T = t_1 \cdots t_n$ of Σ^* is called a *string* or a *text*, where $|T| = n$ denotes its length. Let ϵ be an empty string of length 0, that is, $|\epsilon| = 0$. Let $\Sigma^+ = \Sigma^* \backslash \{\epsilon\}$ and $T = t_1 \cdots t_n \in \Sigma^n$ be any text of length n. If $T = usw$ with $u, s, w \in \Sigma^*$, then s is called a *substring* of T. Let $T[i..j] = t_i \cdots t_j$ for any $1 \leq i \leq j \leq n$ denote a substring of T beginning at i and ending at j in T, and let $T[i] = t_i$ denote the ith symbol of T. For a finite set S of texts, text T is said to be a *superstring* of S if T contains all texts of S as substrings.

Let #occ(s) denote the *frequency* of s, i.e., the number of occurrences of s in a text as a substring. If there exists an isomorphism from an alphabet Σ to another alphabet $\hat{\Sigma}$, texts Σ^* and $\hat{\Sigma}^*$ are said to be *isomorphic* for Σ and $\hat{\Sigma}$.

2.1.2 Maximal Repeats

Let s be a substring of text T. If the frequency of s is greater than 1, s is called a *repeat*. A *left* (or *right*) *extension* of s is any substring of T in a form of ws (or sw), where $w \in \Sigma^*$. We define s as a *left* (or *right*) *maximal* if left (or right) extensions of s occur a strictly less number of times in T than s. Accordingly, s is a *maximal repeat* of T if s is both left and right maximal. In this chapter, we only consider strings with a length of more than 1 as maximal repeats. For example, substring `abra` of T =`abracadabra` is a maximal repeat, whereas `br` is not.

2.1.3 Repetitions and Runs

A *repetition* is a text taking the form w^k, where $w \in \mathcal{A}^+$ and $k \in \mathbb{N}^+$, which means k repetitions of w. A *run* is a repetition satisfying both of the following conditions: (i) $w \in \mathcal{A}$, and (ii) any of its left and right extensions are not repetitions, or the repetition has no left or right extensions.

2.1.4 Bit Encoding Methods for Texts

For a given text, *i-bit encoding* represents each symbol of the text by i bits. *Fixed bit length encoding (FBLE)* represents each symbol by $\lceil \log m \rceil$ bits, where m is the value of the maximum symbol of the text.

A popular algorithm for compact bit encoding of text is *Huffman coding* [42], which

assigns a variable number of bits to text symbols depending on their frequencies and represents each symbol by the assigned number of bits.

Gamma encoding, also known as Elias gamma encoding [33], is an encoding scheme for positive integers. To encode a given number n, gamma encoding inserts $\lfloor \log n \rfloor$ 0s and appends the binary form of n.

Run-length encoding (RLE) converts a given text to two sequences: a *symbol sequence* S and a *length sequence* L. The given text is represented by $r_1 r_2 \cdots r_q$, where $r_i = {a_i}^{k_i}$ with $a_i \in \mathcal{A}$ and $k_i \in \mathbb{N}^+$ for $1 \leq i \leq q$. The obtained S and L are then denoted $a_1 a_2 \cdots a_q$ and $k_1 k_2 \cdots k_q$, respectively. Note that RLE does not generate bit sequences. To complete the bit encoding, we need additional bit-encoding methods for the two sequences generated by RLE.

2.2 Grammars

2.2.1 Context-Free Grammars (CFGs)

A *context-free grammar* (CFG or simply *grammar*) G is defined as a four-tuple $G = \{V, \Sigma, S, R\}$, where V denotes an ordered finite set of *variables*, Σ denotes an ordered finite alphabet, R denotes a finite set of binary relations called *production rules* (or *rules*) between V and $(V \cup \Sigma)^*$, and $S \in V$ denotes a special variable called *start variable*. A production rule refers to the situation, where a variable is substituted and written in a form of $v \to w$, with $v \in V$ and $w \in (V \cup \Sigma)^*$. Let $X, Y \in (V \cup \Sigma)^*$. If there are $x_l, x, x_r, y \in (V \cup \Sigma)^*$ such that $X = x_l x x_r$, $Y = x_l y x_r$, and $x \to y \in R$, we write $X \Rightarrow Y$, and denote the reflexive transitive closure of $\Rightarrow$ as $\overset{*}{\Rightarrow}$. Let $val(v)$ be a string derived from v, i.e., $v \overset{*}{\Rightarrow} val(v)$. We define grammar $\hat{G} = \{\hat{V}, \hat{\Sigma}, \hat{S}, \hat{R}\}$ as a *subgrammar* of G if $\hat{V} \subseteq V$, $\hat{\Sigma} \subseteq (V \cup \Sigma)$, and $\hat{R} \subseteq R$.

2.2.2 Parse-Trees

The *parse tree* of a grammar is a rooted ordered tree with internal nodes labeled by variables and leaves labeled by terminals. Any internal node v_i is related to its children through the rule $v_i \to \alpha_i$, that is, if $\alpha_i = v_{i_1}, v_{i_2}, \cdots, v_{i_j}$ with $j = |\alpha_i|$, the children of v_i are the nodes labeled by $v_{i_1}, v_{i_2}, \cdots, v_{i_j}$ from left to right. Note that the label sequence of the leaves of the parse tree represents the text generated by the grammar.

2.2.3 Run-Length Context-Free Grammars (RLCFGs)

A *run-length context-free grammar* (RLCFG) [45, 76] is an extension of CFG by adding *run-length rules* to the production rules. The run-length rules are written in the form $v \to \alpha^k$ with $\alpha \in (\Sigma \cup V)$ and $k \geq 1$.

2.3 Grammar Compression

2.3.1 Compression using a CFG

Given a text T, *grammar compression* [51] is a method for lossless text data compression that constructs a formal grammar uniquely deriving the text T. Currently, a restricted CFG is mainly used as the grammar. For the CFG to be deterministic, a production rule for each variable $v \in V$ must be unique. In what follows, we assume that every grammar is deterministic and each production rule is $v_i \to \alpha_i$, where α_i is an expression either $\alpha_i = a$ $(a \in \Sigma)$ or $\alpha_i = v_{j_1} v_{j_2} \cdots v_{j_n}$ ($i > j_k$ for all $1 \leq k \leq j_n$).

For estimating the effectiveness of compression, we use the size of the constructed grammar, which is defined as the total length of the right-hand side of all production rules of the grammar.

2.3.2 Compression using a RLCFG

A RLCFG can also be used as the constructed formal grammar in grammar compression. Namely, the grammar impose a unique rule $v_i \to \alpha_i$ on each variable $v_i \in V$ such that α_i is one of $\alpha_i = a$ $(a \in \Sigma)$, $\alpha_i = v_{j_1} v_{j_2} \cdots v_{j_m}$ $(i > j_k$ for all $1 \leq k \leq m)$, or $\alpha_i = {v_j}^k$.

The size of each rule form is specified as follows: (i) Rule $v \to a$ has a size of 1, (ii) rule $v \to v_{j_1} v_{j_2} \cdots v_{j_m}$ has a size of m, and (iii) rule $v \to {v_j}^k$ has a size of 3. We estimate the effectiveness of a compression by the size of its generated grammar, that is, by the total size of its production rules.

2.3.3 RePair algorithm

RePair is a grammar compression algorithm proposed by Larsson and Moffat [56]. For input text T, let $G = \{V, \Sigma, S, R\}$ be the CFG constructed by RePair. Then, the RePair procedure can be described with the following steps:

Step 1. Replace each symbol $a \in \Sigma$ with a new variable v_a and add $v_a \to a$ to R.

Step 2. Find the most frequent pair p in T.

Step 3. Replace every occurrence (or, as many occurrences as possible, when p is a pair consisting of the same symbol) of p with a new variable v, and then, add $v \to p$ to R.

Step 4. Re-evaluate the frequencies of pairs for the updated text generated in **Step 3**. If the maximum frequency is 1, add $S \to$ (current text) to R, and terminate. Otherwise, return to **Step 2**.

Figure 2.1 illustrates an example of the grammar generation process of RePair.

Lemma 1 ([56]). *RePair works in $\mathcal{O}(n)$ expected time and $5n+4k^2+4k'+\lceil\sqrt{n+1}\rceil-1$*

words of space, where n is the length of the source text, k denotes the cardinality of the source alphabet, and k' denotes the cardinality of the final dictionary.

2.4 Model of Computation

2.4.1 Word RAM

In this thesis, we assume the *word RAM* model as the model of computation, that is, a random access machine with a computer word of w bits, where we can carry out all the bitwise operations on a single word in constant time. We ignore the size of names and pointers when we discuss size and space.

	a	b	r	a	c	a	d	a	b	r	a
$v_\alpha \to \alpha$ $(\alpha = \mathtt{a}, \mathtt{b}, \mathtt{r}, \mathtt{c}, \mathtt{d})$	$v_\mathtt{a}$	$v_\mathtt{b}$	$v_\mathtt{r}$	$v_\mathtt{a}$	$v_\mathtt{c}$	$v_\mathtt{a}$	$v_\mathtt{d}$	$v_\mathtt{a}$	$v_\mathtt{b}$	$v_\mathtt{r}$	$v_\mathtt{a}$
$v_1 \to v_\mathtt{a} v_\mathtt{b}$	v_1		$v_\mathtt{r}$	$v_\mathtt{a}$	$v_\mathtt{c}$	$v_\mathtt{a}$	$v_\mathtt{d}$	v_1		$v_\mathtt{r}$	$v_\mathtt{a}$
$v_2 \to v_1 v_\mathtt{r}$	v_2			$v_\mathtt{a}$	$v_\mathtt{c}$	$v_\mathtt{a}$	$v_\mathtt{d}$	v_2			$v_\mathtt{a}$
$v_3 \to v_2 v_\mathtt{a}$	v_3				$v_\mathtt{c}$	$v_\mathtt{a}$	$v_\mathtt{d}$	v_3			
$S \to v_3 v_\mathtt{c} v_\mathtt{a} v_\mathtt{d} v_3$	S										

Figure 2.1: An example of the grammar generation process of RePair for the text `abracadabra`. The generated grammar is $\{\{v_\mathtt{a}, v_\mathtt{b}, v_\mathtt{r}, v_\mathtt{c}, v_\mathtt{d}, v_1, v_2, v_3, S\}, \{\mathtt{a}, \mathtt{b}, \mathtt{r}, \mathtt{c}, \mathtt{d}\}, S, \{v_\mathtt{a} \to \mathtt{a},\ v_\mathtt{b} \to \mathtt{b},\ v_\mathtt{r} \to \mathtt{r},\ v_\mathtt{c} \to \mathtt{c},\ v_\mathtt{d} \to \mathtt{d},\ v_1 \to v_\mathtt{a} v_\mathtt{b},\ v_2 \to v_1 v_\mathtt{c},\ v_3 \to v_2 v_\mathtt{d},\ S \to v_3 v_\mathtt{c} v_\mathtt{a} v_\mathtt{d} v_3\}\}$ with a size of 16.

Chapter 3

Grammar Compression based on Maximal Repeats

This chapter presents an analysis of RePair, which is a grammar compression algorithm known for its simple scheme,while also being practically effective. First, we show that the main process of RePair, that is, the step by step substitution of the most frequent symbol pairs, works within the corresponding most frequent maximal repeats. Then, we reveal the relation between maximal repeats and grammars constructed by RePair. On the basis of this analysis, we further propose a novel variant of RePair, called MR-RePair, which considers the one-time substitution of the most frequent maximal repeats instead of the consecutive substitution of the most frequent pairs. The results of the experiments comparing the size of constructed grammars and execution time of RePair and MR-RePair on several text corpora demonstrate that MR-RePair constructs more compact grammars than RePair does, especially for highly repetitive texts.

3.1 Introduction

Grammar compression is one of the lossless data compression methods. For a given text, grammar compression constructs a small size formal grammar that derives only the given text. Currently, a context-free grammar is mainly used as the formal grammar and in this chapter, we refer to the context-free grammar-based compression as grammar compression. While the problem of constructing the smallest such context-free grammar for a given text is known to be NP-hard [24], several approximation algorithms have been proposed. One of them is RePair [56], which is an off-line grammar compression algorithm. Despite its simple scheme, RePair is known for its high compression in practice [26, 40, 101], and hence, it has been comprehensively studied. Some examples of studies on the RePair algorithm include its extension to an online algorithm [64], practical working time/space improvements [20, 87], applications to various fields [26, 58, 94], and theoretical analysis of generated grammar sizes [24, 71, 77].

In the field of text processing, the repetitiveness of a text is considered an important property. Furthermore, it has been suggested that the extent of the repetitiveness of a given text directly relates to the number of maximal repeats in the text. Belazzougui et al. [13] demonstrated theoretical relations between the number of extensions of maximal repeats and famous other properties of text such as the number of factors in the Lempel–Ziv parsing and the number of runs in the Burrows–Wheeler transform. Also, several text indexing data structures, whose sizes are bounded by the number of extensions of maximal repeats, have been proposed in the literature [11, 12, 98].

In this chapter, we analyzed the properties of RePair, focusing on their relationship to maximal repeats. Although RePair has been extensively studied, to the best of our knowledge, no previous study has associated RePair with maximal repeats. Furthermore, we propose MR-RePair, which is a novel grammar compression algorithm

based on the property of maximal repeats. Several off-line grammar compression techniques based on the properties of (non-maximal) repeats have been proposed previously [10, 44, 67]. Recently, Gańczorz and Jeż introduced a heuristic method that improves the practical compression ratio of RePair in terms of the grammar size [38]. However, none of the previously proposed methods use the properties of maximal repeats. In this chapter, we aim to demonstrate that there is a theoretical guarantee for the size of constructed grammars; under a specific condition, the size of the grammar constructed by MR-RePair is smaller than or equal to that constructed by RePair. Our experiments show that MR-RePair constructs smaller grammars compared to RePair. We emphasize that generating a grammar of small size is of great importance since most, if not all, existing algorithms/data structures that work on grammar-compressed texts have running time dependent on the grammar sizes (see e.g., [27, 28, 39, 41, 16, 43, 46, 17] and the references therein) and not directly on the encoded sizes.

3.1.1 Contributions

The primary contributions of this chapter are as follows.

1. We show interesting relations between maximal repeats and grammars constructed by RePair.

2. We propose MR-RePair, which is a novel variant of RePair based on replacing the most frequent maximal repeats.

3. We implement MR-RePair and experimentally demonstrate that MR-RePair produces smaller grammars than all tested implementations of RePair. For a highly

repetitive text used in the experiments, MR-RePair decreased the size of the constructed grammar to about 55% of that of RePair.

3.1.2 Organization

The rest of this chapter is organized as follows. In Section 3.2, we present an analysis of the properties of RePair and demonstrate its relationship to maximal repeats. The definition and implementation of MR-RePair and its comparison with RePair are provided in Section 3.3. In Section 3.4, we report the experimental results of comparing RePair and MR-RePair. Finally, in Section 3.5, we conclude the chapter.

3.2 Analysis of RePair

This section presents an analysis of RePair with respect to its relationship to maximal repeats and introduces an important concept, called MR-order.

3.2.1 RePair and Maximal Repeats

The following theorem describes an essential property of RePair, that is, RePair recursively replaces the most frequent maximal repeats.

Theorem 1. *Let T be a given text, assuming that every most frequent maximal repeat of T does not appear with overlaps with itself. Let f be the frequency of the most frequent pairs of T, and t be a text obtained after all pairs with frequency f in T are replaced by variables. Then, there is a text s such that s is obtained after all maximal repeats with frequency f in T are replaced by variables, and s and t are isomorphic to each other.*

We need two lemmas and a corollary to prove Theorem 1. The following lemma shows a fundamental relation between the most frequent maximal repeats and the most frequent pairs in a text.

Lemma 2. *A pair p of variables is most frequent in a text T if and only if p occurs once in exactly one of the most frequent maximal repeats of T.*

Proof. ($\Rightarrow$) Let r be a most frequent maximal repeat containing p as a substring. It is clear that p can only occur once in r, since otherwise, $\#\mathrm{occ}(p) > \#\mathrm{occ}(r)$ would hold, implying the existence of a frequent maximal repeat that is more frequent than r, contradicting the assumption that r is most frequent. Suppose that there exists a different most frequent maximal repeat r' containing p as a substring. Similarly, p occurs only once in r'. Furthermore, since r and r' can be obtained by left and right extensions to p, $\#\mathrm{occ}(r) = \#\mathrm{occ}(r') = \#\mathrm{occ}(p)$, and any occurrence of p is contained in an occurrence of both r and r'. Since r' cannot be a substring of r, there exists a string w that is a superstring of r and r', such that $\#\mathrm{occ}(w) = \#\mathrm{occ}(r) = \#\mathrm{occ}(r') = \#\mathrm{occ}(p)$. However, this contradicts that r and r' are maximal repeats.

($\Leftarrow$) Let r be the most frequent maximal repeat such that p occurs once in it. By definition, $\#\mathrm{occ}(r) = \#\mathrm{occ}(p)$. If p is not the most frequent symbol pair in T, there exists a pair p' in T such that $\#\mathrm{occ}(p') > \#\mathrm{occ}(p) = \#\mathrm{occ}(r)$. However, this implies that there is a maximal repeat r' with $\#\mathrm{occ}(r') = \#\mathrm{occ}(p') > \#\mathrm{occ}(r)$, contradicting that r is most frequent. □

The following corollary is derived directly from Lemma 2.

Corollary 1. *For a given text, the frequency of the most frequent pairs and that of the most frequent maximal repeats are the same.*

The following lemma shows an important property of the most frequent maximal repeats.

Lemma 3. *The length of the overlap between any two occurrences of most frequent maximal repeats is at most 1.*

Proof. Let xw and wy be the most frequent maximal repeats that have an overlapping occurrence xwy, where $x, y, w \in \Sigma^+$. If we assume that $|w| \geq 2$, since xw and wy are most frequent maximal repeats, it holds that $\#\mathrm{occ}(w) = \#\mathrm{occ}(xw) = \#\mathrm{occ}(wy)$, i.e., every occurrence of w is preceded by x and followed by y. This implies that $\#\mathrm{occ}(xwy) = \#\mathrm{occ}(xw) = \#\mathrm{occ}(wy)$ as well, but contradicts that xw and wy are maximal repeats. □

Theorem 1 can now be proved based on the above lemmas and corollary.

Proof of Theorem 1. According to Corollary 1, the frequency of the most frequent maximal repeats in T is f. Let p be one of the most frequent pairs in T. According to Lemma 2, there is a unique maximal repeat that is most frequent and contains p once. We denote such maximal repeat as r. Let us assume that there is a substring $zxpyw$ in T, where $z, w \in \Sigma$, $x, y \in \Sigma^*$, and $xpy = r$. We denote $r[1]$ and $r[|r|]$ as $\dot{x}$ and $\dot{y}$, respectively. There are the following two cases to consider:

(i) $\#\mathrm{occ}(z\dot{x}) < f$ **and** $\#\mathrm{occ}(\dot{y}w) < f$**.** If $|r| = 2$, the replacement of p directly corresponds to the replacement of the most frequent maximal repeat, since $p = r$. If $|r| > 2$, after p is replaced with a variable v, r is changed to xvy. This occurs f times in the updated text, and according to Lemma 2, the frequency of every pair occurring in xvy is still f. Because the maximum frequency of pairs does not increase, f is still the maximum frequency. Therefore, we replace all pairs contained in xvy in the following steps, whereas $z\dot{x}$ and $\dot{y}w$ are not replaced. This holds for every occurrence of

p, implying that replacing the most frequent pairs while the maximum frequency does not change, corresponds to replacing all pairs (old and new) contained in the most frequent maximal repeats of the same frequency until they are replaced by a single variable. Then, s can be generated by replacing r.

(ii) $\#\mathrm{occ}(z\dot{x}) = f$ **or** $\#\mathrm{occ}(\dot{y}w) = f$**.** We consider the case where $\#\mathrm{occ}(z\dot{x}) = f$. Note that $\#\mathrm{occ}(zxpy) < f$ according to the assumption that xpy is a maximal repeat. Suppose RePair replaces $z\dot{x}$ by a variable v before p is replaced. Note that according to Lemma 2, there is a maximal repeat occurring f times and including $z\dot{x}$ once (we denote the maximal repeat as r'), and $r' \neq r$ by assumption. According to Lemma 3, the length of the overlap of r and r' is at most 1, and then, only $\dot{x}$ is a symbol present in both r and r'. After that, $xpy = r$ is no longer the most frequent maximal repeat because some of its occurrences are changed to $vr[2..|r|]$. However, $r[2..|r|]$ still occurs f times in the updated text. Since $\#\mathrm{occ}(zxpy) < f$ and $\#\mathrm{occ}(xpy) = f$, $\#\mathrm{occ}(vr[2]) < f$ and $r[2..|r|]$ is a maximal repeat. Then, $r[2..|r|]$ will become a variable in subsequent steps, similarly to **(i)**. Here, r' would also become a variable. Thus, we can generate s by first replacing r' and then replacing $r[2..|r|]$. Similarly, this holds for $\dot{y}w$ when $\#\mathrm{occ}(\dot{y}w) = f$ and $\#\mathrm{occ}(z\dot{x}) = \#\mathrm{occ}(\dot{y}w) = f$. □

3.2.2 MR-Order

According to Theorem 1, if there is just one most frequent maximal repeat in the current text, then RePair replaces its all occurrences step by step. However, a problem arises if there are two or more most frequent maximal repeats, with some of them overlapping. In this case, the selection order of pairs (of course, they are most frequent) affects the priority of maximal repeats. We call this order of selecting (summarizing) maximal repeats as the *maximal repeat selection order* (or simply *MR-order*). Note that, the

selection order of pairs actually depends on the implementation of RePair.

For instance, consider the text `abcdeabccde`, where `abc` and `cde` are the most frequent maximal repeats occurring twice. There are two MR-orders, depending on which of the two maximal repeats `abc` or `cde` is given priority. The results of the replacement using RePair with the MR-order are (i) $xyx\mathtt{c}x$ with variables x and y such that $x \stackrel{*}{\Rightarrow} \mathtt{abc}$ and $y \stackrel{*}{\Rightarrow} \mathtt{de}$, and (ii) $zwz\mathtt{c}w$ with variables z and w such that $z \stackrel{*}{\Rightarrow} \mathtt{ab}$ and $w \stackrel{*}{\Rightarrow} \mathtt{cde}$. More precisely, there are 12 possible ways in which RePair can compress the text, with the following generated rule sets:

1. $\{v_1 \to \mathtt{ab},\ v_2 \to v_1\mathtt{c},\ v_3 \to \mathtt{de},\ S \to v_2v_3v_2\mathtt{c}v_3\}$,
2. $\{v_1 \to \mathtt{ab},\ v_2 \to \mathtt{de},\ v_3 \to v_1\mathtt{c},\ S \to v_3v_2v_3\mathtt{c}v_2\}$,
3. $\{v_1 \to \mathtt{bc},\ v_2 \to \mathtt{a}v_1,\ v_3 \to \mathtt{de},\ S \to v_2v_3v_2\mathtt{c}v_3\}$,
4. $\{v_1 \to \mathtt{bc},\ v_2 \to \mathtt{de},\ v_3 \to \mathtt{a}v_1,\ S \to v_3v_2v_3\mathtt{c}v_2\}$,
5. $\{v_1 \to \mathtt{ed},\ v_2 \to \mathtt{ab},\ v_3 \to v_2\mathtt{c},\ S \to v_3v_1v_3\mathtt{c}v_1\}$,
6. $\{v_1 \to \mathtt{ed},\ v_2 \to \mathtt{bc},\ v_3 \to \mathtt{a}v_2,\ S \to v_3v_1v_3\mathtt{c}v_1\}$,
7. $\{v_1 \to \mathtt{ab},\ v_2 \to \mathtt{cd},\ v_3 \to v_2\mathtt{e},\ S \to v_1v_3v_1\mathtt{c}v_3\}$,
8. $\{v_1 \to \mathtt{ab},\ v_2 \to \mathtt{de},\ v_3 \to \mathtt{c}v_2,\ S \to v_1v_3v_1\mathtt{c}v_3\}$,
9. $\{v_1 \to \mathtt{cd},\ v_2 \to \mathtt{ab},\ v_3 \to v_1\mathtt{e},\ S \to v_2v_3v_2\mathtt{c}v_3\}$,
10. $\{v_1 \to \mathtt{cd},\ v_2 \to v_1\mathtt{e},\ v_3 \to \mathtt{ab},\ S \to v_3v_2v_3\mathtt{c}v_2\}$,
11. $\{v_1 \to \mathtt{ed},\ v_2 \to \mathtt{ab},\ v_3 \to \mathtt{c}v_1,\ S \to v_2v_3v_2\mathtt{c}v_3\}$,
12. $\{v_1 \to \mathtt{ed},\ v_2 \to \mathtt{c}v_1,\ v_3 \to \mathtt{ab},\ S \to v_3v_2v_3\mathtt{c}v_2\}$.

Here, 1–6 have the same MR-order because `abc` precedes `cde` in all of them. At the same time, 7–12 have the same MR-order for the same reason: `cde` precedes `abc`.

If there are several distinct most frequent pairs with overlaps, RePair constructs grammars with different sizes according to the selection order of the pairs. For example, consider the text `bcxdabcyabzdabvbcuda`. There are three most frequent pairs, namely, `ab`, `bc`, and `da`, occurring three times each. If RePair takes `ab` first, the rule set of the generated grammar may become $\{v_1 \to \texttt{ab},\ v_2 \to \texttt{bc},\ v_3 \to \texttt{d}v_1,\ S \to v_2\texttt{x}v_3\texttt{cy}v_1\texttt{z}v_3\texttt{v}v_2\texttt{uda}\}$ and its size is 19. If RePair takes `da` first, the rule set of the generated grammar may become $\{v_1 \to \texttt{da},\ v_2 \to \texttt{bc},\ S \to v_2\texttt{x}v_1v_2\texttt{yabz}v_1\texttt{bv}v_2\texttt{u}v_1\}$ and its size is 18.

Remark 1. *If there are several distinct pairs with the same maximum frequency, the size of the grammar generated by RePair depends on their replacement order.*

However, the following theorem states that the MR-order rather than the replacement order of pairs determines the size of the grammar generated by RePair.

Theorem 2. *The sizes of grammars generated by RePair are the same if they are generated in the same MR-order.*

Proof. Let T be a variable sequence appearing in the grammar generation process of RePair and f be the maximum frequency of pairs in T. Suppose that T' is a variable sequence generated after RePair replaces every pair occurring f times. According to Theorem 1, all generated T' are isomorphic to one another, then the length of all of them is the same, regardless of the replacement order of pairs. Let r_1 be the most frequent maximal repeats of T with r_1 preceding all other maximal repeats in this MR-order. As a result, r_1 is converted into a variable, and according to Lemma 2, all pairs included in r_1 are distinct. Then, the size of the subgrammar which exactly derives

r_1 is $2(|r_1| - 1) + 1 = 2|r_1| - 1$. This holds for the next prioritized maximal repeat (we denote it as r_2) with the following slight difference: the pattern actually replaced would be a substring of r_2 excluding its beginning or end if there are occurrences of overlap with r_1. However, these strings are common in the same MR-order. Then, the sizes of generated subgrammars are the same, regardless of the order of selecting pairs. Similarly, this holds for all most frequent maximal repeats and every maximum frequency of pairs through the entire process of RePair. □

3.2.3 Greatest Size Difference of RePair

We consider the problem of determining the greatest size difference between possible outcomes of RePair.

Definition 1 (Greatest size difference)**.** *Let g and g' be the sizes of any two possible grammars that can be generated by RePair for a given text. Then, the greatest size difference of RePair (GSDRP) is* $\max(|g - g'|)$.

A lower bound of the GSDRP can be established according to the following theorem.

Theorem 3. *Given a text with a length of n, a lower bound of GSDRP is* $\frac{1}{6}(\sqrt{6n+1}+13)$.

Proof. Let B, L, and R be strings such that

$$B = l_1 x y r_1 l_2 x y r_2 \cdots l_{f-1} x y r_{f-1} l_f x y r_f,$$
$$L = \diamondsuit l_1 x \diamondsuit l_2 x \cdots \diamondsuit l_f x,$$
$$R = \diamondsuit y r_1 \diamondsuit y r_2 \cdots \diamondsuit y r_f,$$

where $x, y, l_1, \ldots, l_f, r_1, \ldots, r_f$ denote distinct symbols, and each occurrence of $\diamondsuit$ denotes a distinct symbol. Consider text $T = BL^{f-1}R^{f-1}$. Here, xy, l_1x, $\cdots$, l_fx, yr_1,

$\cdots$, yr_f are the most frequent maximal repeats with a frequency f in T. Let G and G' be grammars generated by RePair for T in different MR-order, such that (i) xy precedes all other maximal repeats and (ii) xy follows all other maximal repeats, respectively. We denote the sizes of G and G' as g and g', respectively.

First, we consider G and how RePair generates it. The first rule generated by the replacement is $v_1 \to xy$ considering the MR-order. After the replacement, L and R remain unchanged, whereas B becomes the following text:

$$B_1 = l_1 v_1 r_1 l_2 v_1 r_2 \cdots l_{f-1} v_1 r_{f-1} l_f v_1 r_f.$$

Each pair in B_1 occurs only once in the entire text $B_1 L^{f-1} R^{f-1}$. This means that B_1 can never be shortened from the current length of $3f$. In the remaining steps, $l_i x$ and yr_i (for $i = 1, \cdots, f$) are replaced. L and R are changed to texts with a length of $2f$ each. Hence, the following holds:

$$g = 3f + 2 \cdot 2f + 2(1 + 2f) = 11f + 2. \tag{3.1}$$

Next, we consider G' and how RePair generates it. According to their MR-order, $l_1 x$, $\cdots$, $l_f x$, yr_1, $\cdots$, yr_f are replaced before xy is selected. They do not overlap with each other, and after they are replaced, xy does not occur in the generated text. Therefore, there are $2f$ rules in G' deriving $l_i x$ and yr_i (for $i = 1, \cdots, f$), whereas the rule deriving xy is absent. L and R are changed to texts with a length of $2f$ each, and B is changed to a text with a length $2f$. Hence, the following holds:

$$g' = 2f + 2 \cdot 2f + 2 \cdot 2f = 10f. \tag{3.2}$$

Let us denote the length of the original text $T = BL^{f-1}R^{f-1}$ by n. Then, the following holds:

$$n = 4f + 2(3f)(f - 1) = 6f^2 - 2f.$$

Therefore,

$$f = \frac{1}{6}(\sqrt{6n+1}+1) \tag{3.3}$$

holds. According to Equations (3.1), (3.2), and (3.3),

$$\begin{aligned} g - g' &= 11f + 2 - 10f = f + 2 \\ &= \frac{1}{6}(\sqrt{6n+1}+13) \end{aligned}$$

holds and the theorem follows. □

3.3 Proposed Method

The main strategy of the proposed method is to recursively replace the most frequent maximal repeats instead of the most frequent pairs.

In this section, we first explain the naïve version of our method called Naïve-MR-RePair. Although it can have a bad performance in certain cases, it is simple and helpful in understanding our main result. Then, we describe the proposed MR-RePair.

3.3.1 Naïve-MR-RePair

Definition 2 (Naïve-MR-RePair)**.** *For an input text T, let $G = \{V, \Sigma, S, R\}$ be the grammar generated by Naïve-MR-RePair. Naïve-MR-RePair constructs G through the following steps:*

Step 1. *Replace each symbol $a \in \Sigma$ with a new variable v_a and add $v_a \to a$ to R.*

Step 2. *Find the most frequent maximal repeat r in T.*

Step 3. *Replace every occurrence (or as many occurrences as possible, when there are overlaps) of r in T with a new variable v and then add $v \to r$ to R.*

Step 4. *Re-evaluate the frequencies of maximal repeats for the updated text generated in* **Step 3**. *If the maximum frequency is 1, add* $S \rightarrow$ (current text) *to* R *and terminate. Otherwise, return to* **Step 2**.

We can easily extend the concept of the MR-order to Naïve-MR-RePair.

Figure 3.1 illustrates an example of the grammar generation process of Naïve-MR-RePair. Figures 2.1 and 3.1 explain why the strategy of using maximal repeats is more effective compared to that using pairs. When compressing the text $v_{\mathtt{a}}v_{\mathtt{b}}v_{\mathtt{r}}v_{\mathtt{a}}v_{\mathtt{c}}v_{\mathtt{a}}v_{\mathtt{d}}v_{\mathtt{a}}v_{\mathtt{b}}v_{\mathtt{r}}v_{\mathtt{a}}$, both RePair and Naïve-MR-RePair generate subgrammars deriving the most frequent maximal repeat $v_{\mathtt{a}}v_{\mathtt{b}}v_{\mathtt{r}}v_{\mathtt{a}}$. The rule set of the subgrammar generated by RePair is $\{v_1 \rightarrow v_{\mathtt{a}}v_{\mathtt{b}}, v_2 \rightarrow v_1v_{\mathtt{r}}, v_3 \rightarrow v_2v_{\mathtt{a}}\}$ with a size of 6. At the same time, the rule set of the subgrammar generated by Naïve-MR-RePair is $\{v_1 \rightarrow v_{\mathtt{a}}v_{\mathtt{b}}v_{\mathtt{r}}v_{\mathtt{a}}\}$ with a size of 4.

However, the following theorem indicates that the size of the grammar generated by Naïve-MR-RePair is larger than that generated by RePair in certain cases, even when they work in the same MR-order. Roughly speaking, this is caused by the overlaps of maximal repeats. When there is an occurrence of the most frequent maximal repeat that overlaps with its occurrence, little difference would arise in grammar constructing processes of RePair and Naïve-MR-RePair from the viewpoint of maximal repeats, that is, the targeted maximal repeats would vary in RePair and in Naïve-MR-RePair (RePair replaces the targeted maximal repeat step by step and Naïve-MR-RePair replaces it at once). Indeed, if maximal repeats are carefully embedded in a text for increasing the difference, the case presented in the following theorem occurs.

Theorem 4. *Given a text* T *with a length of* n, *let* g_{rp} *and* g_{nmr} *be the sizes of the grammars generated by RePair and Naïve-MR-RePair for* T, *respectively, assuming that RePair and Naïve-MR-RePair work in the same MR-order. Then, there is a case*

when $g_{nmr} = g_{rp} + \Omega(\log n)$ *holds.*

Proof. Let $G_{rp} = \{V_{rp}, \Sigma_{rp}, S_{rp}, R_{rp}\}$ and $G_{nmr} = \{V_{nmr}, \Sigma_{nmr}, S_{nmr}, R_{nmr}\}$ be the grammars generated by RePair and Naïve-MR-RePair, respectively. Let T' be the text generated just after **Step 1** of RePair or Naïve-MR-RePair (the **Step 1** is common in both algorithms), that is, $T' = v_1 \cdots v_n$ such that $v_i \in V_{rp} \cap V_{nmr}$ and $v_i \to T[i] \in R_{rp} \cap R_{nmr}$ (for $i = 1, \cdots, n$), and $\hat{G}_{rp} = \{\hat{V}_{rp}, \hat{\Sigma}_{rp}, \hat{S}_{rp}, \hat{R}_{rp}\}$ (or $\hat{G}_{nmr} = \{\hat{V}_{nmr}, \hat{\Sigma}_{nmr}, \hat{S}_{nmr}, \hat{R}_{nmr}\}$) be a subgrammar of G_{rp} (or G_{nmr}) deriving T'. Let $T' = (uw)^{2^{m+1}-1}u$, where $u \in V_{rp} \cap V_{nmr}$, $w \in (V_{rp} \cap V_{nmr})^+$ such that uwu is the most frequent maximal repeat of T', and $m \in \mathbb{N}^+$. Note that $2^{m+1} - 1 = \sum_{i=0}^{m} 2^i$. Here $\hat{R}_{rp}$ and $\hat{R}_{nmr}$ are defined as follows:

$\hat{R}_{rp}$**:** Assume that $x_i \in \hat{V}_{rp}$ for $1 \leq i \leq m$ and $y_j \in \hat{V}_{rp} \cup \hat{\Sigma}_{rp}$ for $1 \leq j \leq |w|$, then $\hat{R}_{rp}$ consists of

- $|w|$ rules $y_j \to y_l y_r$ with $val(y_{|w|}) = uw$,
- one rule $x_1 \to y_{|w|}y_{|w|}$ and $\log_2 \lfloor 2^{m+1} - 1 \rfloor - 1 = m - 1$ rules $x_i \to x_{i-1}x_{i-1}$ for $2 \leq i \leq m$, and
- one rule $\hat{S}_{rp} \to x_m x_{m-1} \cdots x_1 y_{|w|}$.

$\hat{R}_{nmr}$**:** Assume that $d = |\hat{V}_{nmr}| = |\hat{R}_{nmr}|$ and $z_i \in \hat{V}_{nmr}$ for $1 \leq i \leq d$, then $\hat{R}_{nmr}$ consists of

- one rule $z_1 \to uwu$, and
- $d - 1$ rules $z_i \to z_{i-1}wz_{i-1}$ for $2 \leq i \leq d$ and $z_d = \hat{S}_{nmr}$.

Let $\hat{g}_{rp}$ and $\hat{g}_{nmr}$ be the sizes of $\hat{G}_{rp}$ and $\hat{G}_{nmr}$, respectively. Then, the following holds:

$$\hat{g}_{rp} = 2|w| + 2m + (m+2) = 3m + 2|w| + 2, \tag{3.4}$$

$$\hat{g}_{nmr} = |w| + 2 + (|w| + 2)(d-1) = (|w| + 2)d. \tag{3.5}$$

Here, with regard to the length of T', we have

$$n = (2(2^m - 1) + 1)(|w| + 1) + 1, \text{ and}$$
$$n = (2^d - 1)|w| + 2^d.$$

From these, $d = m + 1$ holds. Hence, according to Equations (3.4) and (3.5), the following holds:

$$\hat{g}_{nmr} - \hat{g}_{rp} = (m-1)(|w| - 1) - 1.$$

Therefore, $\hat{g}_{nmr} > \hat{g}_{rp}$ holds for some $(m, |w|)$, and the proposition holds. □

Figures 3.2–3.4 are provided to help in understanding the proof of Theorem 4.

Let G_{rp}, G_{nmr}, and G_{mr} be the grammars generated by RePair, Naïve-MR-RePair, and MR-RePair, respectively. For a given text $T = a_1 \cdots a_n$ ($a_i \in \Sigma,\ 1 \leq i \leq n$) of length $|T| = n$, let g_{rp}, g_{nmr}, and g_{mr} be the sizes of G_{rp}, G_{nmr}, and G_{mr}, respectively. Let us assume that $T = (aw)^{2(2^m-1)+1}a$, where $w \in \Sigma^+$ such that awa is the most frequent maximal repeat of T and $m \in \mathbb{N}^+$. Then, according to the proof of Theorem 4, $g_{nmr} > g_{rp}$ holds for some m and w such that $(m-1)(|w| - 1) > 1$.

Figure 3.2 illustrates a specific example of the grammar generation process of RePair and G_{rp} for $T = (\texttt{abcd})^7\texttt{a}$ with $m = 2$ and $|w| = 3$. The size g_{rp} is 18 in this example. Figure 3.3 illustrates an example of the process of Naïve-MR-RePair and G_{nmr} for the same T. It can be noticed from the figures that the size g_{nmr} is 19, and thus $g_{nmr} > g_{rp}$ holds. As shown in Figure 3.3, Naïve-MR-RePair may fail to extract repetitive patterns

in particular cases (such as $\mathtt{abcd}$ of $(\mathtt{abcd})^7\mathtt{a}$ in the running example). However, this problem can be solved using MR-RePair. Figure 3.4 illustrates an example of the process of MR-RePair and G_{mr} for the same $T = (\mathtt{abcd})^7\mathtt{a}$. The size g_{mr} is 16, which is smaller than $g_{rp} = 18$. Although the most frequent maximal repeat at the second replacement step is $v_{\mathtt{a}}v_{\mathtt{b}}v_{\mathtt{c}}v_{\mathtt{d}}v_{\mathtt{a}}$, MR-RePair replaces $v_{\mathtt{a}}v_{\mathtt{b}}v_{\mathtt{c}}v_{\mathtt{d}}$ with a new variable v_1, providing the additional **Step 3** in Definition 3.

	a	b	r	a	c	a	d	a	b	r	a
$v_\alpha \to \alpha$ $(\alpha = \mathtt{a}, \mathtt{b}, \mathtt{r}, \mathtt{c}, \mathtt{d})$	$v_\mathtt{a}$	$v_\mathtt{b}$	$v_\mathtt{r}$	$v_\mathtt{a}$	$v_\mathtt{c}$	$v_\mathtt{a}$	$v_\mathtt{d}$	$v_\mathtt{a}$	$v_\mathtt{b}$	$v_\mathtt{r}$	$v_\mathtt{a}$
$v_1 \to v_\mathtt{a} v_\mathtt{b} v_\mathtt{r} v_\mathtt{a}$	v_1				$v_\mathtt{c}$	$v_\mathtt{a}$	$v_\mathtt{d}$	v_1			
$S \to v_1 v_\mathtt{c} v_\mathtt{a} v_\mathtt{d} v_1$	S										

Figure 3.1: An example of the grammar generation process of Naïve-MR-RePair for the text `abracadabra`. The generated grammar is $\{\{v_\mathtt{a}, v_\mathtt{b}, v_\mathtt{r}, v_\mathtt{c}, v_\mathtt{d}, v_1, S\}, \{\mathtt{a}, \mathtt{b}, \mathtt{r}, \mathtt{c}, \mathtt{d}\}, S, \{v_\mathtt{a} \to \mathtt{a},\ v_\mathtt{b} \to \mathtt{b},\ v_\mathtt{r} \to \mathtt{r},\ v_\mathtt{c} \to \mathtt{c},\ v_\mathtt{d} \to \mathtt{d},\ v_1 \to v_\mathtt{a} v_\mathtt{b} v_\mathtt{r} v_\mathtt{a},\ S \to v_1 v_\mathtt{c} v_\mathtt{a} v_\mathtt{d} v_1\}\}$ with a size of 14.

<table>
<tr><td></td><td>a</td><td>b</td><td>c</td><td>d</td><td>a</td><td>b</td><td>c</td><td>d</td><td>a</td><td>b</td><td>c</td><td>d</td><td>a</td><td>b</td><td>c</td><td>d</td><td>a</td><td>b</td><td>c</td><td>d</td><td>a</td><td>b</td><td>c</td><td>d</td><td>a</td><td>b</td><td>c</td><td>d</td><td>a</td></tr>
<tr><td>$v_\alpha \rightarrow \alpha\ (\alpha = \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d})$</td><td>$v_\mathtt{a}$</td><td>$v_\mathtt{b}$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td>$v_\mathtt{a}$</td><td>$v_\mathtt{b}$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td>$v_\mathtt{a}$</td><td>$v_\mathtt{b}$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td>$v_\mathtt{a}$</td><td>$v_\mathtt{b}$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td>$v_\mathtt{a}$</td><td>$v_\mathtt{b}$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td>$v_\mathtt{a}$</td><td>$v_\mathtt{b}$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td>$v_\mathtt{a}$</td><td>$v_\mathtt{b}$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td>$v_\mathtt{a}$</td></tr>
<tr><td>$y_1 \rightarrow v_\mathtt{a} v_\mathtt{b}$</td><td colspan="2">$y_1$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td colspan="2">$y_1$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td colspan="2">$y_1$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td colspan="2">$y_1$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td colspan="2">$y_1$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td colspan="2">$y_1$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td colspan="2">$y_1$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td>$v_\mathtt{a}$</td></tr>
<tr><td>$y_2 \rightarrow y_1 v_\mathtt{c}$</td><td colspan="3">$y_2$</td><td>$v_\mathtt{d}$</td><td colspan="3">$y_2$</td><td>$v_\mathtt{d}$</td><td colspan="3">$y_2$</td><td>$v_\mathtt{d}$</td><td colspan="3">$y_2$</td><td>$v_\mathtt{d}$</td><td colspan="3">$y_2$</td><td>$v_\mathtt{d}$</td><td colspan="3">$y_2$</td><td>$v_\mathtt{d}$</td><td colspan="3">$y_2$</td><td>$v_\mathtt{d}$</td><td>$v_\mathtt{a}$</td></tr>
<tr><td>$y_3 \rightarrow y_2 v_\mathtt{d}$</td><td colspan="4">$y_3$</td><td colspan="4">$y_3$</td><td colspan="4">$y_3$</td><td colspan="4">$y_3$</td><td colspan="4">$y_3$</td><td colspan="4">$y_3$</td><td colspan="4">$y_3$</td><td>$v_\mathtt{a}$</td></tr>
<tr><td>$x_1 \rightarrow y_3 y_3$</td><td colspan="8">x_1</td><td colspan="8">x_1</td><td colspan="8">x_1</td><td colspan="4">y_3</td><td>$v_\mathtt{a}$</td></tr>
<tr><td>$x_2 \rightarrow x_1 x_1$</td><td colspan="16">x_2</td><td colspan="8">x_1</td><td colspan="4">y_3</td><td>$v_\mathtt{a}$</td></tr>
<tr><td>$\hat{S}_{rp} \rightarrow x_2 x_1 y_3 v_\mathtt{a}$</td><td colspan="29">$\hat{S}_{rp}$</td></tr>
</table>

Figure 3.2: Grammar generation process of RePair and its generated grammar for the text $(\mathtt{abcd})^7\mathtt{a}$. The grammar size is 18.

<table>
<tr><td></td><td>a</td><td>b</td><td>c</td><td>d</td><td>a</td><td>b</td><td>c</td><td>d</td><td>a</td><td>b</td><td>c</td><td>d</td><td>a</td><td>b</td><td>c</td><td>d</td><td>a</td><td>b</td><td>c</td><td>d</td><td>a</td><td>b</td><td>c</td><td>d</td><td>a</td><td>b</td><td>c</td><td>d</td><td>a</td></tr>
<tr><td>$v_\alpha \rightarrow \alpha\ (\alpha = \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d})$</td><td>$v_\mathtt{a}$</td><td>$v_\mathtt{b}$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td>$v_\mathtt{a}$</td><td>$v_\mathtt{b}$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td>$v_\mathtt{a}$</td><td>$v_\mathtt{b}$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td>$v_\mathtt{a}$</td><td>$v_\mathtt{b}$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td>$v_\mathtt{a}$</td><td>$v_\mathtt{b}$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td>$v_\mathtt{a}$</td><td>$v_\mathtt{b}$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td>$v_\mathtt{a}$</td><td>$v_\mathtt{b}$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td>$v_\mathtt{a}$</td></tr>
<tr><td>$z_1 \rightarrow v_\mathtt{a} v_\mathtt{b} v_\mathtt{c} v_\mathtt{d} v_\mathtt{a}$</td><td colspan="5">$z_1$</td><td>$v_\mathtt{b}$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td colspan="5">$z_1$</td><td>$v_\mathtt{b}$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td colspan="5">$z_1$</td><td>$v_\mathtt{b}$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td colspan="5">$z_1$</td></tr>
<tr><td>$z_2 \rightarrow z_1 v_\mathtt{b} v_\mathtt{c} v_\mathtt{d} z_1$</td><td colspan="13">$z_2$</td><td>$v_\mathtt{b}$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td colspan="13">$z_2$</td></tr>
<tr><td>$\hat{S}_{nmr} \rightarrow z_2 v_\mathtt{b} v_\mathtt{c} v_\mathtt{d} z_2$</td><td colspan="29">$\hat{S}_{nmr}$</td></tr>
</table>

Figure 3.3: Grammar generation process of Naïve-MR-RePair and its generated grammar for the text $(\mathtt{abcd})^7\mathtt{a}$. The grammar size is 19.

<table>
<tr><td></td><td>a</td><td>b</td><td>c</td><td>d</td><td>a</td><td>b</td><td>c</td><td>d</td><td>a</td><td>b</td><td>c</td><td>d</td><td>a</td><td>b</td><td>c</td><td>d</td><td>a</td><td>b</td><td>c</td><td>d</td><td>a</td><td>b</td><td>c</td><td>d</td><td>a</td><td>b</td><td>c</td><td>d</td><td>a</td></tr>
<tr><td>$v_\alpha \rightarrow \alpha\ (\alpha = \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d})$</td><td>$v_\mathtt{a}$</td><td>$v_\mathtt{b}$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td>$v_\mathtt{a}$</td><td>$v_\mathtt{b}$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td>$v_\mathtt{a}$</td><td>$v_\mathtt{b}$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td>$v_\mathtt{a}$</td><td>$v_\mathtt{b}$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td>$v_\mathtt{a}$</td><td>$v_\mathtt{b}$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td>$v_\mathtt{a}$</td><td>$v_\mathtt{b}$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td>$v_\mathtt{a}$</td><td>$v_\mathtt{b}$</td><td>$v_\mathtt{c}$</td><td>$v_\mathtt{d}$</td><td>$v_\mathtt{a}$</td></tr>
<tr><td>$v_1 \rightarrow v_\mathtt{a} v_\mathtt{b} v_\mathtt{c} v_\mathtt{d}$</td><td colspan="4">$v_1$</td><td colspan="4">$v_1$</td><td colspan="4">$v_1$</td><td colspan="4">$v_1$</td><td colspan="4">$v_1$</td><td colspan="4">$v_1$</td><td colspan="4">$v_1$</td><td>$v_\mathtt{a}$</td></tr>
<tr><td>$v_2 \rightarrow v_1 v_1$</td><td colspan="8">v_2</td><td colspan="8">v_2</td><td colspan="8">v_2</td><td colspan="4">v_1</td><td>$v_\mathtt{a}$</td></tr>
<tr><td>$v_3 \rightarrow v_2 v_2$</td><td colspan="16">v_3</td><td colspan="8">v_2</td><td colspan="4">v_1</td><td>$v_\mathtt{a}$</td></tr>
<tr><td>$\hat{S}_{mr} \rightarrow v_3 v_2 v_1 v_\mathtt{a}$</td><td colspan="29">$\hat{S}_{mr}$</td></tr>
</table>

Figure 3.4: Grammar generation process of MR-RePair and its generated grammar for the text $(\mathtt{abcd})^7\mathtt{a}$. The grammar size is 16.

3.3.2 MR-RePair

The grammar size of Naïve-MR-RePair becomes larger than that of RePair as shown in Theorem 4 because Naïve-MR-RePair cannot replace all occurrences of most frequent maximal repeats if it overlaps with another occurrence of itself. In the remainder of this section, we describe MR-RePair, which is an improved version of Naïve-MR-RePair.

Definition 3 (MR-RePair)**.** *For an input text T, let $G = \{V, \Sigma, S, R\}$ be the grammar generated by MR-RePair. MR-RePair constructs T through the following steps:*
Step 1. *Replace each symbol $a \in \Sigma$ with a new variable v_a and add $v_a \to a$ to R.*
Step 2. *Find the most frequent maximal repeat r in T.*
Step 3. *Check if $|r| > 2$ and $r[1] = r[|r|]$, and if so, use $r[1..|r|-1]$ instead of r in* **Step 4**.
Step 4. *Replace every occurrence of r with a new variable v and then add $v \to r$ to R.*
Step 5. *Re-evaluate the frequencies of maximal repeats for the updated text generated in* **Step 4**. *If the maximum frequency is 1, add $S \to$* (current text) *to R and terminate. Otherwise, return to* **Step 2**.

We can easily extend the concept of the MR-order to MR-RePair. We do not care if it uses $r[2..|r|]$ in **Step 3**, instead of $r[1..|r|-1]$. MR-RePair can replace all occurrences of r even if it overlaps with itself in some occurrences since, according to Lemma 3, the length of the overlaps of the most frequent maximal repeats is at most 1. If $r[1] = r[|r|]$ but r does not overlap with itself, then $vr[|r|]$ becomes the most frequent maximal repeat after $r[1..|r|-1]$ is replaced by v and $vr[|r|]$ is replaced immediately. Similar to RePair, MR-RePair still cannot replace all of them if $|r| = 2$.

Figure 3.5 illustrates an example of the grammar generation process of MR-RePair. Although the size of the grammar generated by MR-RePair as shown in Figure 3.5

is larger than that generated by Naïve-MR-RePair as shown in Figure 3.1, it is still smaller than that generated by RePair as shown in Figure 2.1.

Theorem 5. *Assume that RePair and MR-RePair work based on the same MR-order for a given text. Let g_{rp} and g_{mr} be the sizes of the grammars generated by RePair and MR-RePair, respectively. Then, $\frac{1}{2}g_{rp} < g_{mr} \leq g_{rp}$ holds.*

Proof. Assume that $G_{rp} = \{V_{rp}, \Sigma_{rp}, S_{rp}, R_{rp}\}$ and $G_{mr} = \{V_{mr}, \Sigma_{mr}, S_{mr}, R_{mr}\}$ are grammars generated by RePair and MR-RePair, respectively, for a given text T with a length of n. Let T' be the text generated just after **Step 1** of RePair or Naïve-MR-RePair (the **Step 1** is common in both algorithms), that is, $T' = v_1 \cdots v_n$ such that $v_i \in V_{rp} \cap V_{mr}$ and $v_i \to T[i] \in R_{rp} \cap R_{mr}$ (for $i = 1, \cdots, n$).

Let f_1 be the maximum frequency of the maximal repeats in T'. According to Corollary 1, the maximum frequency of the pairs in T' is also f_1. Let $\hat{G}_{rp}^{(f_1)}$ (or $\hat{G}_{mr}^{(f_1)}$) be a subgrammar of G_{rp} (or G_{mr}) generated while RePair (or MR-RePair) replaces pairs (or maximal repeats) with the frequency f_1, $\hat{g}_{rp}^{(f_1)}$ (or $\hat{g}_{mr}^{(f_1)}$) be the size of this subgrammar, and $T_{rp}^{(f_1)}$ (or $T_{mr}^{(f_1)}$) be the updated text after all pairs (or maximal repeats) with the frequency f_1 are replaced. Let $r_1^{(f_1)}, \cdots, r_{m_1}^{(f_1)}$ be maximal repeats with frequency f_1 in T' assuming that they are prioritized in this order by the MR-order. Let $l_i^{(f_1)}$ (for $i = 1, \cdots, m_1$) be the length of the longest substring of $r_i^{(f_1)}$ such that there are variables that derive the substring in both $\hat{G}_{rp}^{(f_1)}$ and $\hat{G}_{mr}^{(f_1)}$. Note that this substring is common to RePair and MR-RePair, and each $l_i^{(f_1)}$ is at least 2. Since RePair replaces such substring step by step and MR-RePair replaces it at once, the

following holds:

$$\hat{g}_{rp}^{(f_1)} = \sum_{i=1}^{m_1} 2(l_i^{(f_1)} - 1) \ , \tag{3.6}$$

$$\hat{g}_{mr}^{(f_1)} = \sum_{i=1}^{m_1} l_i^{(f_1)}. \tag{3.7}$$

From these,

$$\begin{aligned}
\hat{g}_{rp}^{(f_1)} - \hat{g}_{mr}^{(f_1)} &= 2\sum_{i=1}^{m_1} l_i^{(f_1)} - 2m_1 - \sum_{i=1}^{m_1} l_i^{(f_1)} \\
&= \sum_{i=1}^{m_1} l_i^{(f_1)} - 2m_1 \\
&\geq 2m_1 - 2m_1 = 0 \quad (\because l_i \geq 2 \ \text{ holds for each } i).
\end{aligned}$$

Hence,

$$\hat{g}_{mr}^{(f_1)} \leq \hat{g}_{rp}^{(f_1)} \tag{3.8}$$

holds. According to Equation (3.6),

$$\begin{aligned}
\hat{g}_{rp}^{(f_1)} &= 2\sum_{i=1}^{m_1} l_i^{(f_1)} - 2m_1 \\
&= 2\hat{g}_{mr}^{(f_1)} - 2m_1 \quad \text{(by Equation (3.7))}.
\end{aligned}$$

Hence,

$$\frac{1}{2}\hat{g}_{rp}^{(f_1)} < \hat{g}_{mr}^{(f_1)} \tag{3.9}$$

holds. Therefore, according to Equations (3.8) and (3.9),

$$\frac{1}{2}\hat{g}_{rp}^{(f_1)} < \hat{g}_{mr}^{(f_1)} \leq \hat{g}_{rp}^{(f_1)} \tag{3.10}$$

holds. The updated texts $T_{rp}^{(f_1)}$ and $T_{mr}^{(f_1)}$ are isomorphic for V_{rp} and V_{mr}. Let f_2 be the maximum frequency of the maximal repeats in $T_{rp}^{(f_1)}$ (and $T_{mr}^{(f_1)}$). Then, a similar analysis holds for $\hat{G}_{rp}^{(f_2)}$ and $\hat{G}_{mr}^{(f_2)}$. Hence, $\frac{1}{2}\hat{g}_{rp}^{(f_2)} < \hat{g}_{mr}^{(f_2)} \leq \hat{g}_{rp}^{(f_2)}$ holds similarly to Equation (3.10), and the updated texts $T_{rp}^{(f_2)}$ and $T_{mr}^{(f_2)}$ are isomorphic. Inductively, for every maximum frequency of maximal repeats f_i, $\frac{1}{2}\hat{g}_{rp}^{(f_i)} < \hat{g}_{mr}^{(f_i)} \leq \hat{g}_{rp}^{(f_i)}$ holds and the updated texts $T_{rp}^{(f_i)}$ and $T_{mr}^{(f_i)}$ are isomorphic. Let k be a natural number such that $f_k > 1$ and $f_{k+1} = 1$, that is, k is the number of times that the maximum frequency decreases through the entire process of RePair and MR-RePair. Then,

$$\begin{aligned} g_{rp} &= \sum_{j=1}^{k} \hat{g}_{rp}^{(f_j)} + |\Sigma| + |T_{rp}^{(f_k)}| \\ &= \sum_{j=1}^{k} \sum_{i=1}^{m_j} 2(l_i^{(f_j)} - 1) + |\Sigma| + |T_{rp}^{(f_k)}| \ , \quad \text{and} \end{aligned} \tag{3.11}$$

$$\begin{aligned} g_{mr} &= \sum_{j=1}^{k} \hat{g}_{mr}^{(f_j)} + |\Sigma| + |T_{mr}^{(f_k)}| \\ &= \sum_{j=1}^{k} \sum_{i=1}^{m_j} l_i^{(f_j)} + |\Sigma| + |T_{mr}^{(f_k)}| \end{aligned} \tag{3.12}$$

hold. Recall that each symbol $a \in \Sigma$ is replaced with a new variable in the first step both in RePair and in MR-RePair. $|\Sigma|$ is the size of the subgrammar consisting of the rules generated in the first step. Since every $l_i^{(f_j)} \geq 2$ and $|T_{rp}^{(f_k)}| = |T_{mr}^{(f_k)}|$, $\frac{1}{2}g_{rp} < g_{mr} \leq g_{rp}$ follows Equations (3.11) and (3.12), and thus, the proposition holds. $g_{mr} = g_{rp}$ holds when every length $l_i^{(f_j)}$ is 2. □

However, when the MR-orders of RePair and MR-RePair are different, then the grammar generated by MR-RePair can be larger than that generated by RePair, as the following theorem indicates:

Theorem 6. *Unless the MR-order of RePair and MR-RePair are the same, there is a case where the size of the grammar generated by MR-RePair becomes larger than that generated by RePair.*

Proof. We show a concrete example of the case stated in the proposition. Consider text `abcxabcyabczcxcxcycyczcz`. There are four most-frequent maximal repeats, `abc`, `cx`, `cy`, and `cz`. Let A and B be two different MR-orders such that A prioritizes the maximal repeats in order of `abc`, `cx`, `cy`, `cz` and B prioritizes the maximal repeats in order of `cx`, `cy`, `cz`, `abc`, respectively. MR-RePair working in A generates a grammar whose rules are $\{v_{\mathtt{a}} \to \mathtt{a}$, $v_{\mathtt{b}} \to \mathtt{b}$, $v_{\mathtt{c}} \to \mathtt{c}$, $v_{\mathtt{x}} \to \mathtt{x}$, $v_{\mathtt{y}} \to \mathtt{y}$, $v_{\mathtt{z}} \to \mathtt{z}$, $v_1 \to v_{\mathtt{a}}v_{\mathtt{b}}v_{\mathtt{c}}$, $v_2 \to v_{\mathtt{c}}v_{\mathtt{x}}$, $v_3 \to v_{\mathtt{c}}v_{\mathtt{y}}$, $v_4 \to v_{\mathtt{c}}v_{\mathtt{z}}$, $S \to v_1 x v_1 y v_1 z v_2 v_2 v_3 v_3 v_4 v_4\}$, where S is the start variable. Meanwhile, RePair working in B generates a grammar whose rules are $\{v_{\mathtt{a}} \to \mathtt{a}$, $v_{\mathtt{b}} \to \mathtt{b}$, $v_{\mathtt{c}} \to \mathtt{c}$, $v_{\mathtt{x}} \to \mathtt{x}$, $v_{\mathtt{y}} \to \mathtt{y}$, $v_{\mathtt{z}} \to \mathtt{z}$, $v_1 \to v_{\mathtt{c}}v_{\mathtt{x}}$, $v_2 \to v_{\mathtt{c}}v_{\mathtt{y}}$, $v_3 \to v_{\mathtt{c}}v_{\mathtt{z}}$, $v_4 \to v_{\mathtt{a}}v_{\mathtt{b}}$, $S \to v_4 v_1 v_4 v_2 v_4 v_3 v_1 v_1 v_2 v_2 v_3 v_3\}$, where S is the start variable. The size of the grammar generated by MR-RePair working in A is 27, whereas the size of that generated by RePair working in B is 26. □

While Theorem 6 indicates that the grammar can be larger in MR-RePair than in RePair, in Section 3.4 we demonstrate that MR-RePair outperforms RePair in practice.

We can implement MR-RePair by extending the original implementation of RePair stated in [56] and holding the same complexity.

Theorem 7. *Let $G = \{V, \Sigma, S, R\}$ be the grammar generated by MR-RePair for a given text with a length of n. Then, MR-RePair works in $\mathcal{O}(n)$ expected time and $5n + 4k^2 + 4k' + \lceil\sqrt{n+1}\rceil - 1$ word space, where k and k' denote the cardinalities of Σ and V, respectively.*

Proof. Compared to RePair, the additional operations performed by MR-RePair are (i)

extending the selected pair to left and right until it becomes a maximal repeat and (ii) checking and excluding either the beginning or the end of the obtained maximal repeat if they are the same. These additional operations can be realized using the same data structures as those employed in RePair. Then, the space complexity of MR-RePair follows Lemma 1.

We can clearly execute operation (ii) in a constant time. Hence, we consider how the time complexity is affected by operation (i). Let l be the length of the maximal repeat containing the focused pair, as well as f be the frequency of the pair. Then, $\mathcal{O}(fl)$ more time is required for MR-RePair to check the left- and right-extensions for all occurrences of the focused pair compared to RePair. However, the length of the entire text is shortened by at least $f(l-1)$ by the replacement. Therefore, MR-RePair works in $\mathcal{O}(n)$ expected time according to possible counts of the replacement through all of the steps of the algorithm. □

Remark 2. *We can convert a grammar of RePair to that of MR-RePair by repeating the following transform: If a variable v appears only once on the right-hand side of other rules, the rule can be removed for v, and the one occurrence of v can be replaced with the right-hand side of the removed rule. However, the time and space complexity stated in Theorem 7 cannot be achieved in this manner, since additional operations and memory for searching and storing such variables are required.*

3.4 Experiments

We implemented and conducted some comparative experiments. In particular, we compared the sizes of constructed grammars and execution times of the proposed MR-RePair, several existing RePair implementations, and Re-PairImp [37] , which was

recently proposed in [38] as an improvement of RePair.

As stated in Remark 1, the MR-order affects the size of a constructed grammar. In practice, the MR-order varies depending on the implementation of the priority queue that manages pairs. For this reason, we used four different implementations of RePair in the comparative analysis, and they were implemented by Maruyama [61], Navarro [70], Prezza [20, 79], and Wan [102][1], respectively.

Table 3.1 lists the details of the texts that we used in the experiments. In particular, we employed three texts as highly repetitive texts: one is a randomly generated text (rand77.txt), and the other two are a Fibonacci string (fib41) and a German text (einstein.de.txt) selected from the Repetitive Corpus of the Pizza&Chili Corpus [34]. The randomly generated text, rand77.txt, consists of alphanumeric symbols and some special symbols. It was generated by concatenating 32 copies of a block that includes 1024 random strings of length 64 each, i.e., the size of the randomly generated text is $64 \times 1024 \times 32 = 2,097,152$ byte. In addition, we used three texts (E.coli, bible.txt, and world192.txt) selected from the Large Corpus [14], to consider a real-data case. We executed each program seven times for each text and measured the elapsed CPU time only for the grammar generation process. We calculated the average time across five results, excluding the minimum and maximum values among the seven runs. The experiments were run on a computer equipped with an Intel(R) Core i7-8700 3.2-4.6GHz 6core, 32GB RAM, and using Ubuntu 16.04. All of the programs were compiled using gcc version 7.4 with the "-O3" option.

Table 3.2 summarizes the experimental results. Unfortunately, Re-PairImp was unable to process fib41 in our experimental environment because of a lack of memory. Here, we excluded the number of rules generating a single terminal symbol from the

[1] we ran it with level 0 (no heuristic option).

	a	b	r	a	c	a	d	a	b	r	a
$v_\alpha \to \alpha$ ($\alpha = \mathtt{a}, \mathtt{b}, \mathtt{r}, \mathtt{c}, \mathtt{d}$)	$v_\mathtt{a}$	$v_\mathtt{b}$	$v_\mathtt{r}$	$v_\mathtt{a}$	$v_\mathtt{c}$	$v_\mathtt{a}$	$v_\mathtt{d}$	$v_\mathtt{a}$	$v_\mathtt{b}$	$v_\mathtt{r}$	$v_\mathtt{a}$
$v_1 \to v_\mathtt{a} v_\mathtt{b} v_\mathtt{r}$	v_1			$v_\mathtt{a}$	$v_\mathtt{c}$	$v_\mathtt{a}$	$v_\mathtt{d}$	v_1			$v_\mathtt{a}$
$v_2 \to v_1 v_\mathtt{a}$	v_2				$v_\mathtt{c}$	$v_\mathtt{a}$	$v_\mathtt{d}$	v_2			
$S \to v_2 v_\mathtt{c} v_\mathtt{a} v_\mathtt{d} v_2$	S										

Figure 3.5: An example of the grammar generation process of MR-RePair for the text `abracadabra`. The generated grammar is $\{\{v_\mathtt{a}, v_\mathtt{b}, v_\mathtt{r}, v_\mathtt{c}, v_\mathtt{d}, v_1, S\}, \{\mathtt{a}, \mathtt{b}, \mathtt{r}, \mathtt{c}, \mathtt{d}\}, S, \{v_\mathtt{a} \to \mathtt{a},\ v_\mathtt{b} \to \mathtt{b},\ v_\mathtt{r} \to \mathtt{r},\ v_\mathtt{c} \to \mathtt{c},\ v_\mathtt{d} \to \mathtt{d},\ v_1 \to v_\mathtt{a} v_\mathtt{b} v_\mathtt{r},\ v_2 \to v_1 v_\mathtt{a},\ S \to v_2 v_\mathtt{c} v_\mathtt{a} v_\mathtt{d} v_2\}\}$ with a size of 15.

Table 3.1: Text files used in our experiments.

Text	**Size (bytes)**	**$\lvert\Sigma\rvert$**	**Content**
rand77.txt	2,097,152	77	32 copies of 1024 random patterns with a length of 64
fib41	267,914,296	2	Fibonacci string from the Pizza&Chili Corpus
einstein.de.txt	92,758,441	117	Edit history of the Wikipedia for Albert Einstein
E.coli	4,638,690	4	Complete genome of the E. Coli bacterium
bible.txt	4,047,392	63	The King James version of the bible
world192.txt	2,473,400	94	The CIA world fact book

number of rules since they are the same between RePair and MR-RePair. As shown in the table, the sizes of grammars constructed by each RePair implementation differ from each other for all texts except fib41. In any case, MR-RePair is not inferior to RePair in terms of the size of grammars while in Theorem 6 we show that the grammar can be larger in MR-RePair than in RePair if their MR-orders are different. For rand77.txt, the number of rules and size of the grammars for MR-RePair decreased to about 11% and 55% of those for RePair, respectively. Long maximal repeats occur more frequently in rand77.txt than in other texts and we consider this is a main reason of the remarkable effectiveness of MR-RePair for the text.

For einstein.de.txt, the number of rules and size of the grammar decreased to about 44% and 72% of those for RePair, respectively. By contrast, it turned out that the effect of the improvement was limited for the texts from the Large Corpus, which are not highly repetitive. Note that fib41 does not contain any maximal repeats longer than 2 without overlaps. Therefore, MR-RePair generated the same rules as RePair in this case. It should be also be noted that MR-RePair runs at a speed comparable to the fastest implementation of RePair.

3.5 Conclusions

In this chapter, we analyzed the process of RePair and revealed that the RePair algorithm replaces the most frequent pairs step by step within their corresponding most frequent maximal repeats. On the basis of this analysis, we designed MR-RePair, a novel variant of RePair. Instead of consecutively substituting the most frequent pairs, MR-RePair performs one-time substitution of the most frequent maximal repeats. Furthermore, we implemented MR-RePair and compared the sizes of its con-

Table 3.2: Sizes of generated grammars and execution times of the considered algorithms. Each cell in the table represents the number of generated rules, total lengths of the right side of all of the rules except for the start variable, length of the right side of the start variable, and the total grammar size in the order from the top row. The total grammar size presented in the fourth row is the total of the values presented in the second row and the third row. The fifth row separated by a line represents the execution time for compression in seconds. The best results are highlighted in bold.

Text file		RePair Maruyama	Navarro	Prezza	Wan	Re-Pair Imp	MR-RePair
rand77 .txt	Rules	41,651	41,642	41,632	41,675	41,661	**4,492**
	Total length	83,302	83,284	83,264	83,350	83,322	**46,143**
	Start variable	9	**2**	7	**2**	**2**	9
	Grammar size	83,311	83,286	83,271	83,352	83,324	**46,152**
	Execution time	0.22	0.34	2.94	0.94	2.48	**0.20**
fib41	Rules	38	38	38	38	-	38
	Total length	76	76	76	76	-	76
	Start variable	**3**	**3**	**3**	**3**	-	**3**
	Grammar size	**79**	**79**	**79**	**79**	-	**79**
	Execution time	**9.99**	14.38	48.85	85.39	-	14.88
einstein .de.txt	Rules	49,968	49,949	50,218	50,057	49,933	**21,787**
	Total length	99,936	99,898	100,436	100,114	99,866	**71,709**
	Start variable	12,734	12,665	13,419	**12,610**	12,672	12,683
	Grammar size	112,670	112,563	113,855	112,724	112,538	**84,392**
	Execution time	**9.04**	13.74	136.49	40.24	213.73	9.73
E.coli	Rules	66,664	66,757	66,660	67,368	66,739	**62,363**
	Total length	133,328	133,514	133,320	134,736	133,478	**129,138**
	Start variable	651,875	**649,660**	650,538	652,664	650,209	650,174
	Grammar size	785,203	783,174	783,858	787,400	783,687	**779,312**
	Execution time	**0.52**	0.65	9.82	2.00	11.29	0.58
bible.txt	Rules	81,193	81,169	80,999	81,229	81,282	**72,082**
	Total length	162,386	162,338	161,998	162,458	162,564	**153,266**
	Start variable	386,514	386,381	386,992	386,094	**385,989**	386,516
	Grammar size	548,900	548,719	548,990	548,552	548,553	**539,782**
	Execution time	**0.51**	0.65	8.41	1.85	11.32	0.57
world 192.txt	Rules	55,552	55,798	55,409	55,473	55,437	**48,601**
	Total length	111,104	111,596	110,812	110,946	110,874	**104,060**
	Start variable	213,131	213,962	213,245	**212,647**	212,857	212,940
	Grammar size	324,235	325,558	324,057	323,593	323,731	**317,000**
	Execution time	**0.32**	0.55	4.92	1.09	6.81	0.36

structed grammars to those of the grammars constructed by several implementations of RePair. Through the experiments, we confirmed the effectiveness of MR-RePair especially for highly repetitive texts.

We defined the greatest size difference of any two possible grammars that can be generated by RePair for a given text, naming it GSDRP. We demonstrated that a lower bound of GSDRP is $\frac{1}{6}(\sqrt{6n+1}+13)$ for a given text of length n.

We estimated the effectiveness of the compression using the size of the generated grammars instead of the length of the output bits. Reducing the grammar size has important implications since the majority of the existing text algorithms applied to grammar-compressed texts, including grammar-based self indexes [27, 28], edit distance computation [39], q-gram mining [41, 16], and pattern matching [43, 46, 17], have time/space complexities that are dependent on the input grammar size. For instance, the compressed indexes proposed by Claude and Navarro [27, 28] can be directly built on MR-RePair grammar-compressed texts. Algorithms specifically designed for *straight-line programs* (*SLPs*), which are text compressions with grammars in Chomsky normal form, can also be easily modified to work on grammars that are not in Chomsky normal form similar to MR-RePair grammars. Hence, MR-RePair serves as a base for practical improvements of these algorithms.

From the viewpoint of storing data more compactly, developing a method for encoding constructed grammars is another important issue. We discuss implementing an efficient encoding method for MR-RePair in Chapter 4.

Chapter 4

Grammar Compression with Run-Length Rules

Grammar compression aims to construct a small-sized context-free grammar (CFG) that uniquely generates the input text data. Theoretically, the effectiveness of CFG compression can be improved by run-length CFG (RLCFG), an extension of CFG. However, compression algorithms have been proposed only for the CFG scheme; compression algorithms for the RLCFG scheme have not been studied so the practical compression effectiveness of this scheme remains undiscovered. Here, we design a practical compression algorithm for the RLCFG scheme and a compact bit-encoding method for the constructed RLCFG. In experimental evaluations on real repetitive datasets, we demonstrate the high performance of the compression scheme based on the proposed algorithm and the encoding method.

4.1 Introduction

Grammar compression is a lossless data compression method that constructs a small-sized formal grammar that uniquely derives the input text. In grammar compression, a context-free grammar (CFG) is mainly used as the formal grammar. As generating the smallest such CFG from a given text is NP-hard [24], it is achieved by various approximation techniques.

Run-length CFG (RLCFG) is an extension of CFG applied by Jeż [45] but formally introduced by Nishimoto et al. [76]. The theoretical properties of RLCFG were studied in [18, 36]. Theoretically, RLCFG improves the effectiveness of CFG compression, but whether this is realized in practice remains undiscovered because, to our knowledge, no compression algorithms for the RLCFG scheme have been developed.

RePair [56], an off-line grammar compression algorithm for the CFG scheme, achieves a high compression ratio [26, 40, 101] despite its simple scheme. RePair has attracted considerable interest and has been extended to an online algorithm [64], embellished with practical working time/space improvements [20, 87], applied to other fields [26, 58, 94], and subjected to theoretical analysis of its generated grammar sizes [24, 71, 77]. In Chapter 3, we proposed a variant of RePair called MR-RePair. The experimental results showed that the grammars were smaller in MR-RePair than in the original RePair for repetitive datasets; that is, the compression efficiency was higher in MR-RePair than in RePair.

In this chapter, we design a novel compression algorithm for the RLCFG-based grammar compression scheme called RL-MR-RePair, which follows RePair and MR-RePair for the CFG scheme. Experimentally, we show that RL-MR-RePair constructs smaller grammars for repetitive datasets than either RePair or MR-RePair.

Generating small-sized grammars is undeniably important since grammar-compressed

texts can be processed by several algorithms and data structures, whose runtimes depends on the grammar size [39, 41, 16, 43, 46]. Meanwhile, these grammars must be encoded in the storage format of compressed data, namely, as compact bit sequences.

Related to RePair, succinct encoding of straight-line programs (SLP) was addressed in [95] (indeed, the grammars constructed by RePair are easily transformed to SLPs). In addition, Bille et al. [19, 79] proposed a variant of RePair and an effective coding method for the variant algorithm. However, encoding methods for MR-RePair have not been discussed. Without effective encoding methods, the final bit sequence of the grammar might be larger in MR-RePair than in RePair, even if the grammar is smaller in MR-RePair than in RePair.

In this chapter, we also propose a bit-encoding method for constructed RLCFGs. The scheme was originally designed for RL-MR-RePair but is directly applicable to MR-RePair. In comparative experiments from grammar construction to final bit encoding, we evaluate the performance of RePair, MR-RePair, and RL-MR-RePair. The experiments confirm the high compression performance of RL-MR-RePair with the proposed encoding method on real repetitive datasets.

4.1.1 Contributions

The primary contributions of this chapter are listed below.

1. We design a new compression algorithm for the RLCFG scheme called RL-MR-RePair.

2. We propose an encoding scheme for RL-MR-RePair and MR-RePair.

3. We implement RePair, MR-RePair and RL-MR-RePair and experimentally confirm that RL-MR-RePair produces smaller grammars than the other methods in

nearly all instances. Furthermore, we implement eight encoding methods for Re-Pair and six encoding methods for MR-RePair and RL-MR-RePair and confirm their compression effectiveness in comparative experiments.

4.1.2 Organization

The remainder of this chapter is organized as follows. Section 4.2 describes the RL-MR-RePair algorithm and its implementation and analyzes its time and space complexities, Section 4.3 introduces some encoding schemes for grammar compression and presents our bit-encoding method, Section 4.4 is dedicated experimental results, and Section 4.5 concludes the chapter.

4.2 Proposed Method

In Chapter 3, we proposed MR-RePair as a variant of RePair that identifies not the most frequent pair, but the most frequent maximal repeat. We reported that MR-RePair more efficiently compresses the grammar size than RePair in practice and is especially effective for repetitive data. In this section, we extend MR-RePair to run-length grammar compression schemes and present a new variant of RePair called RL-MR-RePair.

4.2.1 Algorithm

Let x be a symbol and k be a natural number such that $k \geq 2$. The most frequent maximal repeat of x^k is x^2. Conversely, if the most frequent maximal repeat in T is x^2, then T might contain a long repetition of x^k. RL-MR-RePair searches run x^k in T

and replaces that run if the most frequent maximal repeat is x^2. Otherwise, it works similarly to MR-RePair. The RL-MR-RePair algorithm is given as Algorithm 1.

4.2.2 Implementation

RL-MR-RePair is implemented similarly to MR-RePair but with an additional hash the replacement phase (cf., Line 9 in Algorithm 1) to check whether a run has occurred previously. If the run has already occurred, the replacement phase uses the previous variable; otherwise, a new variable is required.

Theorem 8. *RL-MR-RePair executes in $\mathcal{O}(n)$ expected time, where n is the length of the input text.*

Proof. Unlike MR-RePair, RL-MR-RePair must check whether a run is a repeat run and requires an additional operation for each replacement of the old variable with a new one. Assuming that the extra hash works in $\mathcal{O}(1)$ expected time, the time complexity of RL-MR-RePair equals to that of MR-RePair; that is, by Theorem 7, RL-MR-RePair executes in $\mathcal{O}(n)$ expected time. □

Theorem 9. *For a given text of length n, let us denote the grammar constructed by RL-MR-RePair by $\{\Sigma, V, s, R\}$. Then, the space complexity of RL-MR-RePair is $6n + 4|\Sigma|^2 + 4V + \lceil\sqrt{n+1}\rceil - 1$ words.*

Proof. RL-MR-RePair requires one more space than MR-RePair for the extra hash, which maintains the length of the runs occurring in the text. As the total length of such runs is at most n, the hash requires at most n words of space. By Theorem 7, RL-MR-RePair requires $6n + 4|\Sigma|^2 + 4V + \lceil\sqrt{n+1}\rceil - 1$ words of space. □

Algorithm 1 RL-MR-RePair

Input: T

Output: $G = \{V, \Sigma, s, R\}$

Replace each $a \in \Sigma$ in T with a new variable v_a and then
add v_a to V and $v_a \to a$ to R.
loop
    Find the most frequent maximal repeat r.
    if $\#\mathrm{occ}(r) < 2$ **then**
        Add $s \to T$ to R.
        return G
    end if
    if $r = x^2$ with variable x **then**
        Replace each run x^k with a new variable v_k and then
        add v_k to V and $v_k \to x^k$ to R.
    else
        if $|r| > 2$ and $r[1] = r[|r|]$ **then**
            $r \leftarrow r[1..|r| - 1]$
        end if
        Replace each r in T with a new variable v and then
        add v to V and $v \to r$ to R.
    end if
end loop

4.3 Bit Encoding

Let $G = \{\Sigma, V, s, R\}$ be a grammar constructed by RePair, MR-RePair, or RL-MR-RePair, where $\Sigma = \{a_1, \cdots, a_\sigma\}$, $V = \{1, \cdots, (\sigma + d + 1)\}$, $s = (\sigma + d + 1)$, and $R = \{1 \to a_1, \cdots, \sigma \to a_\sigma, (\sigma + 1) \to \alpha_1, \cdots, (\sigma + d) \to \alpha_d, (\sigma + d + 1) \to \tau\}$. In the following discussion, the right-hand side of each run-length rule $v_i \to {v_j}^k$ is written as a symbol sequence $0kv_j$, where 0 is a special symbol implying that the expression is the right-hand side of a run-length rule. In this representation, the RLCFG is treated as a CFG.

Finally, the compressed data are stored as bit sequences. In the most simple encoding approach, G is converted to a text, which is then encoded by a general text-encoding scheme such as i-bit encoding, FBLE, or Huffman coding (e.g., the RePair implementation by Navarro [70] uses 32-bit encoding). Here, we can convert G to $a_1 \cdots a_\sigma \diamond \alpha_1 \diamond \alpha_2 \diamond \cdots \diamond \alpha_d \diamond \tau$, where $\diamond$ is a special symbol called a *delimiter*. Letting g be the size of G, the length of the resulting text is $\sigma + 1 + \sum_i^d (|\alpha_i| + 1) + |\tau| = g + d + 1$. If the length of each α_i is 2, we can reduce the number of delimiters by converting G to $a_1 \cdots a_\sigma \diamond \alpha_1 \alpha_2 \cdots \alpha_d \diamond \tau$ with length $\sigma + 1 + \sum_i^d |\alpha_i| + 1 + |\tau| = g + 2$. This implies that the final bit sequence of the grammar can be smaller in RePair than in MR-RePair or RL-MR-RePair, even if the grammar is larger in RePair than in MR-RePair or RL-MR-RePair.

4.3.1 A Previous Effective Method for RePair

Bille et al. [19, 79] proposed another variant of RePair and an effective encoding for the variant algorithm. They partially sorted the grammar rules and encoded the grammar by *packed gamma encoding (PGE)*, defined as follows.

Definition 4 (PGE)**.** *Given a text T and a natural number ε, let D be a sequence such that $\lceil \log l_1 \rceil \lceil \log l_2 \rceil \cdots \lceil \log l_q \rceil$, where l_i is the value of the maximum symbol in $T[j..j+\varepsilon-1]$ with $j = \varepsilon(i-1)$ and $q = \lfloor |T|/\varepsilon \rfloor$. Also, let D_{delta} be a sequence with first entry $D_{delta}[1] = D[1]+1$ and remaining entries $D_{delta}[i] = |D[i] - D[i-1]| + 1$ for $1 < i \leq q$, and let D_{pms} be a bit sequence with first entry $D_{pms}[1] = 1$. For the remaining entries $1 < i \leq q$, if $D[i] \geq D[i-1]$, then $D_{pms}[i] = 1$; otherwise, $D_{pms}[i] = 0$. Let S_1 and L_1 be the symbol sequence and length sequence obtained by RLE of D_{delta}, respectively. Similarly, S_2 and L_2 are the symbol and length sequences obtained by RLE of L_1, respectively. Then, the* PGE *encodes T as a bit sequence consisting of the following five-bit sequences.*

1. *A gamma-encoded bit sequence of S_1.*
2. *A gamma-encoded bit sequence of S_2.*
3. *A gamma-encoded bit sequence of L_2.*
4. *A bit sequence obtained by representing each symbol $T[i]$ in T by $D[\lfloor i/\varepsilon \rfloor]$ bits with $1 \leq i \leq |T|$.*
5. *D_{pms}.*

PGE is expected to perform well when symbols in a text have similar values to their adjacent symbols. Bille et al. [19, 79] applied PGE to RePair as follows; (i) construct two texts X and X_{delta} such that $X = \max(\alpha_i[1], \alpha_i[2])$ and $X_{delta} = |\alpha_i[1] - \alpha_i[2]|$ (note that the length of each α_i in RePair is 2), (ii) construct a bit sequence X_{pms} such that if $X[i]$ is $\alpha_i[1]$ then $X_{pms}[i] = 1$; otherwise, $X_{pms}[i] = 0$, and (iii) store X_{delta}, X_{pms}, and the PGE-encoded bit sequence of X. In this encoding scheme, each α_i must be of length 2, which is unsuitable for either MR-RePair or RL-MR-RePair.

4.3.2 Encoding via Post-Order Partial Parse Tree (POPPT)

A *partial parse tree* [81] is an ordered tree formed by traversing the parse tree in a depth-first manner and pruning out all descendants under each node of variables appearing at least twice. A *POPPT* [62] is a partial parse tree whose internal nodes contain post-order variables. A *post-order CFG (POCFG)* [62] is a CFG whose partial parse tree is a POPPT. For the compact encoding of general grammars (not only CFGs with $|\alpha_i| = 2$ for each i), a succinct representation of POCFG is useful.

Takabatake et al. [97][1] encoded POCFG as a succinct representation comprising a bit sequence B and a text U. B is built by traversing the partial parse tree P (specifically, a POPPT) of the POCFG in post-order and assigning c 0s and one 1 to a node with c children. Finally, a single 0 is inserted in B to represent the super node. The text U stores the symbols of the leaves of P from left to right.

Takabatake et al. [97] mentioned a similar encoding for SLPs developed by Maruyama et al. [62]. The SLP encoding similarly constructs a bit sequence B and a text U but builds B by traversing the POPPT in post-order and inserting 0 in B if the node is a leaf, or 1 otherwise. The constructed B is smaller than that of Takabatake et al. [97]. This method is easily applied to the grammar constructed by RePair with τ decomposed as $\{(\sigma+d+1) \to \tau[1]\tau[2], (\sigma+d+2) \to (\sigma+d+1)\tau[3], \cdots, (\sigma+d+|\tau|-1) \to (\sigma+d+|\tau|-2)\tau[|\tau|]\}$. This decomposition increases the size of the grammar by $|\tau|$, but does not affect the final representation since the variables $(\sigma+d+1), \cdots, (\sigma+d+|\tau|-1)$ do not explicitly appear and only the symbols occurring in τ are placed in U.

[1] To construct a rank/select dictionary, the leaves of the partial parse tree in [97] are marked by an additional bit sequence. We omit this procedure because our method does not use the bit sequence.

4.3.3 Combination of POPPT and PGE

Both methods outlined in Section 4.3.2 finally encode U as a bit sequence. In these methods, each symbol $U[i]$ in the encoded U is represented by $\lceil \log (i + |\Sigma|) \rceil$ bits for $1 \leq i \leq |U|$. This representation is referred to as *increasing bit length encoding (IBLE)*. Here, note that $U[i] \leq i + |\Sigma|$ holds since in a POPPT, the value of a leaf node is at most the number of internal nodes in post-order until the leaf node is reached.

As an alternative method for encoding U, we propose a scheme based on PGE. We expect that PGE will properly encode U since there is a tendency that the values of the symbols in U are close to those of their adjacent symbols.

4.4 Experiments

This section experimentally compare the performance of RePair, MR-RePair, and RL-MR-RePair. The measured value were the sizes of the grammars, the execution times and peak memory usages of the grammar constructions, the sizes of the final compressed files, and the execution times of encoding the grammars. The tested datasets are listed in Table 4.1. All datasets were obtained from the Repetitive Corpus produced in Pizza&Chili Corpus [34]. All tests were conducted on an Intel(R) Core i7-7800 X 3.50 GHz 12-core with 64 GB RAM. The operating system was Linux (Ubuntu 16.04.2, 64bit) running kernel 4.15.0. All programs were compiled by `rustc` version 1.35.0 with the `--release` option.

4.4.1 Grammar Construction

Table 4.2 shows the sizes of the grammars constructed by RePair, RePair(PS), MR-RePair, and RL-MR-RePair, the execution time of each grammar construction, and

the peak memory usage of each grammar construction. RePair(PS) is a variant of the RePair developed in [19][2], which partially sorts the grammar rules in the procedure. While we stated in Theorem 6 that there is a case in which MR-RePair is theoretically inferior to RePair, MR-RePair and RL-MR-RePair generally outperformed RePair and RePair(PS). In particular, RL-MR-RePair constructed the smallest grammars on all datasets except `coreutils` and executed faster than MR-RePair on all datasets except sources.001.2.

4.4.2 Encoding the Grammars

Tables 4.3, 4.4, and 4.5 show the sizes of the files compressed by RePair, MR-RePair, and RL-MR-RePair, respectively. We tested eight encoding methods by RePair and six encoding methods by MR-RePair and RL-MR-RePair. The tables give the execution times of the encoding methods.

`32bit`, `fble`, and `huffman` convert a given grammar to text by the procedure introduced in the exordium of Section 4.3 and encode the text by 32-bit encoding, FBLE, and Huffman coding, respectively. Note that the number of delimiters in the converted text is smaller in RePair than in the other methods. `32bit` is the simplest conversion procedure and directly reflects the size of the grammar. However, as shown in the tables, 32bit is too large for symbol representation. Also, `huffman` tended to be larger than `fble`. We consider that Huffman coding performed poorly because few symbols occurred repeatedly in the converted text.

For comparison, we also encoded RePair (more precisely, RePair(PS)) by the method

[2]A program of the algorithm implemented by the authors is available in [79]. For a fair comparison, we tested our own implementation of the algorithm (the implementation in [79] mainly aims to reduce the memory usage, which slightly decreases the runtime performance).

of Bille et al. [19, 79]. The implementation in [79] sets the constant ε to 6 (see Definition 4), but we found that setting $\varepsilon = 8$ improves the efficiency of the compression in some cases; thus, we evaluated both $\varepsilon = 6$ (`ps+pge6`) and $\varepsilon = 8$ (`ps+pge8`). As shown in Table 4.3, both ε settings significantly improved the compression efficiency over methods that convert a given grammar to a text.

Finally, we tested POPPT-based encoding methods for every RePair variant. The POPPT for RePair was constructed by the method of Maruyama et al. [62], and those for MR-RePair and RL-MR-RePair were constructed by the method of Takabatake et al. [97]. In the succinct representation of POCFG, we expressed the text in three ways; IBLE (`poppt+ible`), PGE with $\varepsilon = 6$ (`poppt+pge6`), and PGE with $\varepsilon = 8$ (`poppt+pge8`). The POPPT-based methods achieved high compression efficiency in MR-RePair and RL-MR-RePair; in particular, `poppt+pge8` achieved the best compression ratio on all datasets except fib41 and para (in para, `poppt+pge6` delivered the best performance, folloed by `poppt+pge8`). As shown in Table 4.3, the POPPT-based methods were effective even on the grammars constructed by RePair.

Table 4.6 summarizes the best compression performances achieved by each RePair variant in above experiments. For comparison, we also show the compression results of two famous file compressors, gzip (version 1.6, with the `-9` option) and bzip2 (version 1.0.6, with the `-9` option). As indicated in the table, RePair effectively compressed the artificial datasets (A) and the pseudo-real datasets (PR). In contrast, MR-RePair and RL-MR-RePair performed well on the real datasets (R), and RL-MR-RePair achieved higher compression efficiency than MR-RePair on all datasets except coreutils.

4.5 Conclusions

We extended the MR-RePair algorithm to a run-length context-free grammar and designed a novel variant called RL-MR-RePair. In addition, we proposed an encoding scheme for MR-RePair and RL-MR-RePair and experimentally compared the performances of the RePair variants on real repetitive datasets. The experimental results confirmed the high compression performance of RL-MR-RePair and the proposed encoding scheme.

Table 4.1: Datasets used in the experiments. Here, $|\Sigma|$ is the alphabet size, representing the number of different symbols occurring in each dataset. The "type" describes the scheme used to generate the dataset: artificially created symbol sequences (A), artificially generated by adding repetitiveness to real data (PR), or real repetitive data (R).

Name	Size (bytes)	$\|\Sigma\|$	Type	Description
fib41	267,914,296	2	A	Fibonacci string
dna.001.1	104,857,600	5	PR	100 × 1MiB prefix of human genome
sources.001.2	104,857,600	98	PR	100 × 1MiB prefix of Linux and GCC sources
coreutils	205,281,778	236	R	9 versions of GNU Coreutils source
einstein.en.txt	467,626,544	139	R	Edit history of Wikipedia for Albert Einstein
influenza	154,808,555	15	R	78,041 DNA sequences of Haemophilus Influenzae
para	429,265,758	5	R	36 DNA sequences of Saccharomyces Paradoxus
world_leaders	46,968,181	89	R	CIA World Leaders from Jan. 2003 to Dec. 2009

Table 4.2: Sizes of the generated grammars, execution times, and peak memory usages of the grammar constructions. In each constructed grammar $G = \{\Sigma, V, s, R\}$, $\Sigma = \{a_1, \cdots, a_\sigma\}$, $V = \{1, \cdots, (\sigma + d + 1)\}$, $s = (\sigma + d + 1)$, and $R = \{1 \to a_1, \cdots, \sigma \to a_\sigma, (\sigma + 1) \to \alpha_1, \cdots, (\sigma + d) \to \alpha_d, (\sigma+d+1) \to \tau\}$. From the top row, each cell represents d, $(\sum_{i=0}^{d} |\alpha_i|)$, $|\tau|$, and the size of grammar G. The fifth and sixth rows (separated by a line) state the average runtimes of five executions (in seconds) and the average peak memory usages of the five executions (in kB), respectively. The best results are highlighted in bold font.

	RePair	RePair(PS)	MR-RePair	RL-MR-RePair
fib41	**38**	**38**	**38**	**38**
	76	**76**	**76**	**76**
	3	**3**	**3**	**3**
	81	**81**	**81**	**81**
	67.163	67.183	81.341	81.162
	18,122,200	18,122,220	18,921,276	18,921,412
dna.001.1	261,023	261,239	223,983	**223,612**
	522,046	522,478	485,514	**485,251**
	498,612	498,402	496,566	**494,406**
	1,020,663	1,020,885	982,085	**979,662**
	61.787	61.798	70.493	67.244
	7,684,652	7,685,988	**7,660,588**	7,660,980
sources.001.2	709,174	709,052	400,258	**400,213**
	1,418,348	1,418,104	1,109,686	**1,109,548**
	183,656	183,583	181,393	**181,253**
	1,602,102	1,601,785	1,291,177	**1,290,899**
	64.770	64.924	68.147	69.077
	7,708,468	7,708,868	7,620,056	**7,619,772**
coreutils	1,833,094	1,833,918	436,515	**436,443**
	3,666,188	3,667,836	**2,269,133**	2,269,393
	154,036	154,001	153,622	**153,611**
	3,820,460	3,822,073	**2,422,991**	2,423,240
	122.824	124.309	137.012	128.219
	15,529,364	15,529,764	15,226,140	**15,225,932**
einstein.en.txt	100,681	100,641	49,373	**49,221**
	201,362	201,282	**150,105**	150,173
	62,492	62,580	62,318	**62,096**
	263,993	264,001	212,562	**212,408**
	294.028	**293.325**	323.260	320.619
	25,181,396	25,181,724	24,741,612	**24,735,416**
influenza	659,560	659,473	427,595	**423,419**
	1,319,120	1,318,946	1,088,157	**1,077,405**
	897,431	898,010	894,544	**887,131**
	2,216,566	2,216,971	1,982,716	**1,964,551**
	87.705	87.819	103.473	98.723
	13,240,848	13,242,928	13,228,356	**13,109,972**
para	3,076,152	3,077,085	**1,079,287**	1,082,467
	6,152,304	6,154,170	4,157,167	**4,145,790**
	1,142,696	1,142,356	1,134,361	**1,121,371**
	7,295,005	7,296,531	5,291,533	**5,267,166**
	248.267	249.652	279.212	260.333
	32,160,672	32,165,360	**31,602,712**	31,603,136
world_leaders	209,071	209,079	99,910	**98,078**
	418,142	418,158	309,031	**306,091**
	98,127	98,210	97,712	**94,851**
	516,358	516,457	406,832	**401,031**
	20.214	20.427	23.553	**18.586**
	4,222,924	4,222,713	**4,163,532**	4,164,368

Table 4.3: Sizes of the files compressed by RePair and the execution times of encoding the grammars. In each cell, the first row represents the size (bytes), and the second row (in parentheses) gives the compression ratio (compressed file size)/(input file size)×100 (%). The third row (separated by a line) is the average runtime of five executions (in seconds).

	RePair							
	`32bit`	`fble`	`huffman`	`ps+pge6`	`ps+pge8`	`poppt+ible`	`poppt+pge6`	`poppt+pge8`
fib41	327	85	251	83	84	**50**	69	71
	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)	**(0.0000)**	(0.0000)	(0.0000)
	0.000	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**
dna	4,082,646	2,296,506	3,211,576	1,783,251	**1,778,453**	1,957,954	1,906,365	1,911,272
.001.1	(3.8935)	(2.1901)	(3.0628)	(1.7006)	**(1.6961)**	(1.8673)	(1.8181)	(1.8277)
	0.085	0.119	0.627	**0.078**	**0.078**	0.156	0.166	0.162
sources	6,408,123	4,005,269	6,713,711	2,606,298	2,604,665	**2,324,485**	2,342,488	2,334,697
.001.2	(6.1113)	(3.8197)	(6.4027)	(2.4856)	(2.4840)	**(2.2168)**	(2.2340)	(2.2265)
	0.148	0.214	1.438	0.169	0.167	0.272	0.286	0.283
coreutils	15,281,141	10,028,722	17,020,682	5,655,042	5,657,054	**5,451,520**	5,469,724	5,461,667
	(7.4440)	(4.8853)	(8.2914)	(2.7548)	(2.7558)	**(2.6556)**	(2.6645)	(2.6606)
	0.329	0.533	3.867	0.420	0.394	0.688	0.714	0.710
einstein	1,055,564	560,999	938,266	439,083	441,650	375,523	**374,902**	374,938
.en.txt	(0.2257)	(0.1200)	(0.2006)	(0.0939)	(0.0944)	(0.0803)	**(0.0802)**	(0.0802)
	0.024	0.031	0.165	0.027	0.027	0.037	0.039	0.038
influenza	8,866,228	5,541,429	7,813,734	4,140,255	**4,137,727**	4,214,266	4,201,129	4,198,308
	(5.7272)	(3.5795)	(5.0474)	(2.6744)	**(2.6728)**	(2.7222)	(2.7138)	(2.7119)
	0.189	0.295	1.603	0.203	0.199	0.387	0.409	0.402
para	29,180,014	20,061,278	30,566,314	11,812,763	11,893,263	12,135,356	**11,710,363**	11,759,392
	(6.7977)	(4.6734)	(7.1206)	(2.7519)	(2.7498)	(2.8270)	**(2.7280)**	(2.7394)
	0.631	1.053	7.193	0.723	0.712	1.423	1.460	1.455
world_	2,065,174	1,161,820	1,953,121	796,540	796,666	741,111	740,316	**739,570**
leaders	(4.3970)	(2.4736)	(4.1584)	(1.6959)	(1.6962)	(1.5779)	(1.5762)	**(1.5746)**
	0.046	0.061	0.380	0.049	0.049	0.078	0.081	0.081

Table 4.4: As for Table 4.3, but showing the size of the files compressed by MR-RePair and the execution times of encoding the grammars.

	MR-RePair					
	`32bit`	`fble`	`huffman`	`poppt+ible`	`poppt+pge6`	`poppt+pge8`
fib41	429	118	264	**60**	79	78
	(0.0000)	(0.0000)	(0.0000)	**(0.0000)**	(0.0000)	(0.0000)
	0.000	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**
dna.001.1	4,824,266	2,713,667	3,073,145	1,918,499	1,895,294	**1,894,870**
	(4.6008)	(2.5880)	(2.9308)	(1.8296)	(1.8075)	**(1.8071)**
	0.096	0.139	0.560	0.144	0.151	0.149
sources.001.2	6,765,455	4,017,172	4,791,544	2,373,197	2,343,157	**2,335,164**
	(6.4520)	(3.8311)	(4.5696)	(2.2633)	(2.2346)	**(2.2270)**
	0.145	0.217	0.933	0.215	0.219	0.217
coreutils	11,437,333	6,791,346	7,368,357	5,258,079	5,115,689	**5,106,577**
	(5.5715)	(3.3083)	(3.5894)	(2.5614)	(2.4920)	**(2.4876)**
	0.222	0.363	1.427	0.393	0.403	0.394
einstein.en.txt	1,047,332	523,884	626,349	371,338	363,071	**362,624**
	(0.2240)	(0.1120)	(0.1339)	(0.0794)	(0.0776)	**(0.0775)**
	0.021	0.027	0.096	0.026	0.028	0.028
influenza	9,641,208	5,724,503	6,427,285	4,123,574	4,071,746	**4,064,247**
	(6.2278)	(3.6978)	(4.1518)	(2.6637)	(2.6302)	**(2.6253)**
	0.198	0.313	1.253	0.326	0.346	0.351
para	25,483,274	16,723,417	16,887,956	12,117,901	**11,269,822**	11,306,815
	(5.9365)	(3.8958)	(3.9341)	(2.8229)	**(2.6254)**	(2.6340)
	0.502	0.884	3.565	1.009	1.019	1.005
world_leaders	2,026,710	1,076,841	1,275,874	737,552	719,313	**717,965**
	(4.3151)	(2.2927)	(2.7165)	(1.5703)	(1.5315)	**(1.5286)**
	0.041	0.057	0.226	0.058	0.056	0.056

Table 4.5: As for Table 4.3, but showing the size of the files compressed by RL-MR-RePair and the execution times of encoding the grammars.

	RL-MR-RePair					
	`32bit`	`fble`	`huffman`	`poppt+ible`	`poppt+pge6`	`poppt+pge8`
fib41	479	118	264	**60**	79	78
	(0.0000)	(0.0000)	(0.0000)	**(0.0000)**	(0.0000)	(0.0000)
	0.000	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**
dna.001.1	4,813,090	2,707,381	3,067,205	1,913,276	1,889,730	**1,889,630**
	(4.5901)	(2.5820)	(2.9251)	(1.8246)	(1.8022)	**(1.8021)**
	0.096	0.144	0.551	0.144	0.150	0.150
sources.001.2	6,764,163	4,016,405	4,789,737	2,372,574	2,342,202	**2,334,317**
	(6.4508)	(3.8303)	(4.5678)	(2.2627)	(2.2337)	**(2.2262)**
	0.141	0.215	0.929	0.208	0.217	0.216
coreutils	11,438,025	6,791,756	7,367,012	5,258,904	5,115,868	**5,106,824**
	(5.5719)	(3.3085)	(3.5887)	(2.5618)	(2.4921)	**(2.4877)**
	0.222	0.359	1.330	0.381	0.395	0.393
einstein.en.txt	1,046,112	523,274	624,555	371,298	362,760	**362,372**
	(0.2237)	(0.1119)	(0.1336)	(0.0794)	(0.0776)	**(0.0775)**
	0.022	0.027	0.097	0.026	0.028	0.028
influenza	9,551,844	5,671,443	6,367,819	4,085,808	4,033,158	**4,025,295**
	(6.1701)	(3.6635)	(4.1134)	(2.6393)	(2.6053)	**(2.6002)**
	0.192	0.301	1.223	0.318	0.334	0.329
para	25,398,526	16,667,801	16,876,589	12,039,499	**11,203,814**	11,240,382
	(5.9167)	(3.8829)	(3.9315)	(2.8047)	**(2.6100)**	(2.6185)
	0.501	0.884	3.364	0.933	0.958	0.952
world_leaders	1,996,178	1,060,621	1,253,963	727,668	708,711	**707,450**
	(4.2501)	(2.2582)	(2.6698)	(1.5493)	(1.5089)	**(1.5062)**
	0.040	0.057	0.200	0.055	0.056	0.056

Table 4.6: Sizes of the files compressed by the gzip, bzip2, and RePair variants (the methods achieving the highest compression performance are highlighted in bold font). From top to bottom in each row (datasets) are listed the size (bytes), compression ratio (compressed file size)/(input file size)×100 (%), and the encoding method.

	gzip	bzip2	RePair	MR-RePair	RL-MR-RePair
fib41	1,176,257	14,893	**50**	60	60
	(0.4390)	(0.0056)	**(0.0000)**	(0.0000)	(0.0000)
			poppt+ible	poppt+ible	poppt+ible
dna.001.1	28,486,029	27,385,893	**1,778,453**	1,894,870	1,889,630
	(27.1664)	(26.1172)	**(1.6961)**	(1.8071)	(1.8021)
			ps+pge8	poppt+pge8	poppt+pge8
sources.001.2	36,023,271	34,619,138	**2,324,485**	2,335,164	2,334,317
	(34.3545)	(33.0154)	**(2.2168)**	(2.2270)	(2.2262)
			poppt+ible	poppt+pge8	poppt+pge8
coreutils	49,920,838	32,892,028	5,451,520	**5,106,577**	5,106,824
	(24.3182)	(16.0229)	(2.6556)	**(2.4876)**	(2.4877)
			poppt+ible	poppt+pge8	poppt+pge8
einstein.en.txt	163,664,285	24,157,362	374,902	362,624	**362,372**
	(34.9989)	(5.1660)	(0.0802)	(0.0775)	**(0.0775)**
			poppt+pge6	poppt+pge8	poppt+pge8
influenza	10,636,889	10,197,176	4,137,727	4,064,247	**4,025,295**
	(6.8710)	(6.5870)	(2.6728)	(2.6253)	**(2.6002)**
			ps+pge8	poppt+pge8	poppt+pge8
para	116,073,220	112,233,085	11,710,363	11,269,822	**11,203,814**
	(27.0399)	(26.1454)	(2.7280)	(2.6254)	**(2.6100)**
			poppt+pge6	poppt+pge6	poppt+pge6
world_leaders	8,287,665	3,260,930	739,570	717,965	**707,450**
	(17.6453)	(6.9428)	(1.5746)	(1.5286)	**(1.5062)**
			poppt+pge6	poppt+pge8	poppt+pge8

Chapter 5

Compaction of Natural Numbers for Higher-Order Compression

In this chapter, we address the problem of compaction of Church numerals. Church numerals are unary representations of natural numbers on the scheme of lambda terms. We propose a novel decomposition scheme from a given natural number into an arithmetic expression using tetration, which enables us to obtain a compact representation of lambda terms that leads to the Church numeral of the natural number. For natural number n, we prove that the size of the lambda term obtained by the proposed method is $\mathcal{O}((\mathrm{slog}_2 n)^{(\log n/\log\log n)})$. Moreover, we experimentally confirmed that the proposed method outperforms binary representation of Church numerals on average, when n is less than approximately 10,000.

5.1 Introduction

The goal of this chapter is to obtain a compact lambda term (λ-term) that leads to the Church numeral of a given natural number. Church numerals are unary representations of natural numbers on λ-terms. Let $\mathcal{C}(n)$ be the Church numeral for natural number n; then $\mathcal{C}(n)$ is as follows:

$$\mathcal{C}(n) = \lambda f.\lambda x.\overbrace{f\ (f\ \cdots\ (f}^{n}\ x)\cdots).$$

Namely, the length of the Church numeral increases linearly with n. We want a more compact representation for n in the scheme of λ-terms. We refer to this task as the *compaction of Church numerals*. Using a binary representation for n is a natural idea, which enables to obtain a λ-term whose size is $\Theta(\log n)$. Another idea is to decompose n into an arithmetic expression and to represent it as a corresponding λ-term of the expression. This may reduce the size of the λ-term. For example, $n = 500$ can be decomposed into $5 \times 10 \times 10$. The λ-term corresponding to this expression is given as $(\lambda p.\lambda q.\lambda f.\lambda x.p\ (q\ (q\ f))\ x)\ \mathcal{C}(5)\ \mathcal{C}(10)$, which is much smaller than $\mathcal{C}(500)$. In this idea, *tetration* can be used for the decomposition. Tetration is the next hyperoperation after exponentiation; it is defined as iterated exponentiation, e.g., the third tetration of two ${}^3 2 = 2^{2^2} = 16$. Since a function for calculating ${}^i k$ on Church numerals is written as a λ-term whose size is $\mathcal{O}(i + k)$, the size of the λ-term for $n = {}^i k$ achieves $o(\log n)$.

In this chapter, we propose the *Recursive tetrational partitioning* (RTP) method to decompose a natural number using tetration. We also present an algorithm to generate a compact λ-term by using RTP. Moreover, we prove that the size of the obtained λ-term is $\mathcal{O}((\mathrm{slog}_2 n)^{\log n/\log\log n})$, where $\mathrm{slog}_2 n$ is super-logarithm, which is the inverse operation of tetration. It is slightly worse than $\mathcal{O}(\log n)$, but we also prove that the size of the λ-term achieves $\mathcal{O}(\mathrm{slog}_2 n)$ in some cases, and our experimental result shows

that our method outperforms a binary representation on λ-term on average, when n is less than approximately 10,000.

The compaction of Church numerals can be applied to data compression. Kobayashi et al. [53] proposed a compression method called higher-order compression that uses extended λ-terms as the data model. Their method translates an input data to a compact λ-term and then encodes the obtained λ-term. When a pattern consecutively appears n times in the input, such a part can be represented as a λ-term with $\mathcal{C}(n)$. Thus efficient compaction of $\mathcal{C}(n)$ will help for higher-order compression when the data contains many repetitive patterns.

Yaguchi et al. [104] proposed an efficient algorithm for higher-order compression. They utilized a simply typed λ-term for efficient modeling and encoding. Differing from Kobayashi et al.'s method where each context occurring more than once in an input is extracted, Yaguchi et al.'s method extracts the most frequent context up to a certain size. They state in [104] that their method often achieves better performance than a grammar compression [51] with regard to compression ratio. Of course, their method can also be applied to the compaction of $\mathcal{C}(n)$, while it takes time for repeating context extraction. We confirmed via experiments that our method tends to produce more compact λ-terms for Church numerals than Yaguchi et al.'s method. Note that our method can be easily incorporated into Yaguchi et al.'s method.

5.1.1 Contributions

The primary contributions of this chapter are as follows.

1. For natural numbers, we propose a novel decomposition scheme called RTP, which enables to obtain a compact representation of λ-terms that leads to the Church numerals of the numbers.

2. By incorporating RTP, we propose an algorithm to perform the compaction of $\mathcal{C}(n)$ for natural number n. Moreover, we prove that the size of the λ-terms constructed by the algorithm is $\mathcal{O}((\mathrm{slog}_2 n)^{\log n/\log\log n})$.

3. We implemented the proposed algorithm and conducted comparative experiments. With regard to the sizes of the obtained λ-terms, the results show that the method is superior to that of Yaguchi et al. [104] and is also superior to a binary representation on λ-term on average, when n is less than approximately $10,000$.

5.1.2 Organization

The rest of this chapter is organized as follows. In Section 5.2, we review tetration, lambda notation, and Church numerals. In Section 5.3, we define the proposed RTP method and present the translation algorithm using RTP. We also prove the upper bound of the size of the λ-term produced by our algorithm. In Section 5.4, we discuss application to higher-order compression and present our experimental result. Conclusions are presented in Section 5.5.

5.2 Preliminaries

This section provides the definitions and notations used in this chapter. We introduce tetration and super-logarithm and we review λ-terms and Church numerals.

5.2.1 Tetration and Super-Logarithm

Definition 5 (Tetration)**.** *For natural numbers φ and t, the t-th tetration of φ, denoted by ${}^t\varphi$, is recursively defined as follows:*

$$ {}^t\varphi := \begin{cases} 1 & (\textit{if } \ t = 0), \\ \varphi^{({}^{t-1}\varphi)} & (\textit{otherwise}). \end{cases} $$

For example, ${}^1 2 = 2^{{}^0 2} = 2^1 = 2$, ${}^2 2 = 2^{{}^1 2} = 2^2 = 4$, ${}^3 2 = 2^{{}^2 2} = 2^4 = 16$, and ${}^4 2 = 2^{{}^3 2} = 2^{16} = 65536$.

The following corollary is easily induced from Definition 5.

Corollary 2. *For natural numbers φ and t, it holds that $\log_\varphi {}^t\varphi = {}^{t-1}\varphi$.*

Definition 6 (Super-logarithm)**.** *Super-logarithm, denoted by* slog*, is the inverse operation of tetration, defined as $\mathrm{slog}_\varphi({}^t\varphi) = t$ with natural numbers φ and t.*

The *iterated logarithm* of n, denoted by $\log^* n$, is also known as the number of times the logarithm function must be applied to n before it becomes less than or equal to 1. For positive numbers, the super-logarithm is essentially equivalent to the iterated logarithm, i.e., it holds that $\log^* n = \lceil \mathrm{slog}_e n \rceil$ for any $n > 0$. However, note that in the $\mathcal{O}(.)$-notation the size of super-logarithm depends on its base, that is, $\mathcal{O}(\mathrm{slog}_a n) \neq \mathcal{O}(\mathrm{slog}_b n)$ if $a \neq b$.

5.2.2 Lambda Terms

Definition 7 (Lambda terms (λ-terms))**.** *Let x be a variable. Then, lambda term (λ-term) M is defined inductively as follows:*

$$ M ::= x \mid \lambda x.M \mid M_1 \ M_2. $$

We call λ-terms formed $\lambda x.M$ and $M_1\ M_2$ λ-*abstraction* and *functional application*, respectively. Let V_1 and V_2 be the sets of the variables occurring in M_1 and M_2, respectively, then, we identify M_1 with M_2 if there is an isomorphism from V_1 to V_2. $M[x_1 := M_1, x_2 := M_2, \dots]$ denotes the λ-term such that all occurrences of $x_1, x_2, \dots$ in M are replaced by $M_1, M_2, \dots$, respectively.

We show in parentheses "(" and ")" the precedence of combining λ-terms. However, for simplicity, we omit these parentheses based on standard omission rules, that is, we consider that functional applications are left-associative and prior to λ-abstractions. For example, we consider $M_1\ M_2\ M_3$ as $(M_1\ M_2)\ M_3$ rather than $M_1\ (M_2\ M_3)$, and consider $\lambda x.M_1 M_2$ as $\lambda x.(M_1\ M_2)$ rather than $(\lambda x.M_1)\ M_2$.

Definition 8 (β-reduction)**.** *Let x be a variable, and M_1 and M_2 be λ-terms. Then, β-*reduction *is a relation of λ-terms, denoted by using $\longrightarrow_\beta$, such that $(\lambda x.M_1)\ M_2 \longrightarrow_\beta M_1[x := M_2]$. We write $\longrightarrow_\beta^*$ for the reflexive transitive closure of $\longrightarrow_\beta$.*

For example, $(\lambda x.M_1\ x\ x)\ M_2 \longrightarrow_\beta M_1\ M_2\ M_2$ because the right-side of $\longrightarrow_\beta$ is $(M_1\ x\ x)[x := M_2]$. Similarly, $(\lambda x.\lambda y.x\ y\ y)\ M_1\ M_2 \longrightarrow_\beta^* M_1\ M_2\ M_2$ because $(\lambda x.\lambda y.x\ y\ y)\ M_1\ M_2 \longrightarrow_\beta (\lambda y.M_1\ y\ y)\ M_2 \longrightarrow_\beta M_1\ M_2\ M_2$.

We define the sizes of λ-terms as follows. The definition can be found in [53].

Definition 9 (Sizes of λ-terms)**.** *Let x be a variable and M be a λ-term. Then, we denote the size of the λ-term by $\#M$, and inductively define the size of each λ-term as follows:*

$$\#x = 1, \quad \#(\lambda x.M) = \#M + 1, \quad \#(M_1\ M_2) = \#M_1 + \#M_2 + 1.$$

For λ-terms M_1 and M_2, we say that M_2 is more *compact* than M_1 if $\#M_2 < \#M_1$. *Compaction* of a λ-term M_1 means to find a λ-term M_2 such that $M_1\ N \longrightarrow_\beta^* M'$ and $M_2\ N \longrightarrow_\beta^* M'$ for every N with a λ-term M', and $\#M_2 < \#M_1$.

In this chapter, we assume Word RAM as the computational model. Thus, we ignore the size of names and pointers when we discuss size and space.

5.2.3 Church Numerals

It is known that natural numbers are represented as *Church numerals* on λ-terms. Also, some λ-terms enabling arithmetic operations on Church numerals are provided.

Definition 10 (Church numerals)**.** *For natural number n, its Church numeral, denoted by $\mathcal{C}(n)$, is*

$$\mathcal{C}(n) := \lambda f.\lambda x.\overbrace{f\ (f \cdots (f}^{n}\ x) \cdots).$$

Corollary 3 (Arithmetic operations on Church numerals)**.** *Let n_1 and n_2 be natural numbers. Then we obtain three λ-terms $\mathrm{Add}(n_1, n_2)$, $\mathrm{Mul}(n_1, n_2)$, and $\mathrm{Tet}(n_1, n_2)$, such that $\mathrm{Add}(n_1, n_2) \longrightarrow_\beta^* \mathcal{C}(n_1 + n_2)$, $\mathrm{Mul}(n_1, n_2) \longrightarrow_\beta^* \mathcal{C}(n_1 \cdot n_2)$, and $\mathrm{Tet}(n_1, n_2) \longrightarrow_\beta^* \mathcal{C}({}^{n_2}n_1)$, respectively, as follows:*

$$\begin{aligned}
\mathrm{Add}(n_1, n_2) &= (\lambda p.\lambda q.\lambda f.\lambda x.p\ f\ (q\ f\ x))\ \mathcal{C}(n_1)\ \mathcal{C}(n_2), \\
\mathrm{Mul}(n_1, n_2) &= (\lambda p.\lambda q.\lambda f.\lambda x.p\ (q\ f)\ x)\ \mathcal{C}(n_1)\ \mathcal{C}(n_2), \\
\mathrm{Tet}(n_1, n_2) &= (\lambda p.\lambda f.\lambda x.\overbrace{p\ p \cdots p}^{n_2}\ f\ x)\ \mathcal{C}(n_1).
\end{aligned}$$

As can be seen in Corollary 3, in each λ-term, one λ-abstraction occurs first, and one or two Church numerals follow it. We call the former λ-abstraction and the following Church numerals *function term* and *argument terms*, respectively. For example, for $\mathrm{Add}(n_1, n_2)$, $(\lambda p.\lambda q.\lambda f.\lambda x.p\ f\ (q\ f\ x))$ is the function term and the following $\mathcal{C}(n_1)$ and $\mathcal{C}(n_2)$ are the argument terms.

5.2.4 Binary Expression of Natural Numbers on λ-Terms

Instead of Church numerals, we can represent natural numbers on λ-terms based on binary representation as follows [66].

Definition 11 (Binary representation on λ-terms)**.** *For natural number n, assume that its binary representation is $B_k B_{k-1} \cdots B_2 B_1$ with $B_i \in \{0, 1\}$ for $1 \leq i \leq k$. Then, the binary expression of n on λ-terms, denoted by $\mathcal{B}(n)$, is*

$$\mathcal{B}(n) := b_{B_1}\ (b_{B_2}\ \cdots\ (b_{B_{k-1}}\ (b_{B_k}\ (\lambda x.x)))\cdots),$$

where $b_0 = \lambda p.\lambda f.\lambda x.p\ f\ (p\ f\ x)$ and $b_1 = \lambda p.\lambda f.\lambda x.f\ (p\ f\ (p\ f\ x))$.

For example, the binary expression of 57 is 111001, then its binary expression on λ-terms is $b_1\ (b_0\ (b_0\ (b_1(\ b_1\ (b_1\ (\lambda x.x))))))$.

Note that $\mathcal{B}(n) \longrightarrow_\beta^* \mathcal{C}(n)$ and $\#\mathcal{B}(n) = \Theta(\log n)$ follows Definition 11, while $\#\mathcal{C}(n) = \Theta(n)$.

5.3 Proposed Method

5.3.1 Basic Idea

Let n be a natural number. As stated in Section 5.1, we can reduce $\#\mathcal{C}(n)$ by using decomposition of n into an arithmetic expression. For the example, for $n = 500$, we can decompose it to an arithmetic expression $5 \cdot 10 \cdot 10$ (more precisely, $(5 \cdot 10) \cdot 10$). A corresponding λ-term M of the expression is obtained as $M = (\lambda p.\lambda q.\lambda f.\lambda x.p\ (q\ f)\ x)\ ((\lambda p.\lambda q.\lambda f.\lambda x.p\ (q\ f)\ x)\ \mathcal{C}(5)\ \mathcal{C}(10))\ \mathcal{C}(10)$, and its size is 85, while $\#\mathcal{C}(500) = 1003$. Here, $M \longrightarrow_\beta^* \mathcal{C}(500)$. Moreover, we can make M more compact by combining two λ-abstractions into a single λ-abstraction, such as

$(\lambda p.\lambda q.\lambda f.\lambda x.p\ (q\ (q\ f))\ x)\ \mathcal{C}(5)\ \mathcal{C}(10)$. This also generates $\mathcal{C}(500)$ through β-reduction similar to M, while its size is just 51.

We perform compaction of $\mathcal{C}(n)$ in the following three steps.

Step 1. Decompose n into a special form of arithmetic expression, called TAE.

Step 2. Translate the arithmetic expression into a λ-term having the form of $M\ \mathcal{C}(\varphi^*)$.

Step 3. Apply the above **Step 1** and **Step 2** for the natural number φ^* recursively.

For **Step 1**, we introduce TAEs in Section 5.3.2. For **Step 2**, we describe a translation method in Section 5.3.3. There are many ways to achieve decomposition of a natural number into a TAE, and the size of the translated λ-term changes depending on the expression. Then, we consider how we effectively obtain a TAE such that the translated λ-term of the TAE is compact. We propose RTP in Section 5.3.4, which is a heuristic approach decomposing a natural number into a TAE. Finally, for **Step 3**, we discuss further recursive compaction of Church numerals in Section 5.3.5.

5.3.2 Tetrational Arithmetic Expression (TAE)

We define *tetrational arithmetic expression (TAE)*, denote by E, as a special form of arithmetic expression.

Definition 12 (Tetrational arithmetic expressions (TAEs))**.** *Let N be an arbitrary natural number. Then, tetrational arithmetic expression (TAE) E is inductively defined as follows:*

$$E ::= N \mid E + E \mid E \cdot E \mid {}^{N_2}N_1.$$

If the evaluated result of a TAE E is a natural number n, we write $E[n]$ for the TAE. Especially, let E_φ denote a restricted form of TAE with a natural number φ,

defined as

$$E_\varphi ::= \varphi \mid E_\varphi + E_\varphi \mid E_\varphi \cdot E_\varphi \mid {}^t\varphi,$$

where t is an arbitrary natural number.

For example, $2 \cdot 3 + {}^2 2$ is a TAE, which is written as $E[10]$ because its evaluated result is equal to 10. Similarly, ${}^2 2 \cdot 2 + 2$ is also a TAE written as $E_2[10]$.

Informally speaking, reducing the number of natural numbers occurring in TAEs is effective for compaction of corresponding λ-terms, since it enables to reduce the number of the kinds of argument terms in the λ-terms; note that we can transform $(\lambda p.\lambda q.\lambda f.\lambda x.p\ f\ (q\ f\ x))\ \mathcal{C}(n_1)\ \mathcal{C}(n_1)$ to $(\lambda p.\lambda f.\lambda x.p\ f\ (p\ f\ x))\ \mathcal{C}(n_1)$, for example. From this point of view, it is expected that the corresponding λ-term of $E_\varphi[n]$ becomes compact when n is a multiple of φ. Let $r = n \bmod \varphi$ and $\bar{n} = n - r$. Thus, we consider TAE $E_\varphi[\bar{n}] + r$ instead of $E_\varphi[n]$ itself.

5.3.3 Translation from TAE to λ-Term

Here, we consider how we translate an TAE $E_\varphi[\bar{n}]$ to a corresponding λ-term. A simple method is to translate each arithmetic operation occurring in $E_\varphi[\bar{n}]$ by using λ-terms stated in Corollary 3. For example, for $E_2[200] = {}^3 2 \cdot ({}^2 2 \cdot 2 + {}^2 2) + {}^2 2 \cdot 2$, the translated λ-term is obtained as

$$\mathrm{Add}(\mathrm{Mul}(\mathrm{Tet}(2,3), \mathrm{Add}(\mathrm{Mul}(\mathrm{Tet}(2,2),2), \mathrm{Tet}(2,2))), \mathrm{Mul}(\mathrm{Tet}(2,2),2)).$$

As seen above, the corresponding λ-term seems to become long. However, we can translate the TAE to a compact λ-term.

Definition 13 (Corresponding functional λ-terms (CFLT))**.** *Let n and φ be natural numbers. We say a λ-term M is a corresponding functional λ-term (CFLT) of TAE $E[n]$ if both of the following conditions hold for M:*

1. *M has the form of $(\lambda p.\lambda f.\lambda x.\hat{M})\ \mathcal{C}(\varphi)$, where $\hat{M}$ is a λ-term that contains no Church numerals.*

2. *$M \longrightarrow_\beta^* \mathcal{C}(n)$.*

For a given TAE E, We denote a CFLT of E by $\Lambda(E)$.

For example, for $E_2[200] = {}^3 2 \cdot ({}^2 2 \cdot 2 + {}^2 2) + {}^2 2 \cdot 2$, one of its CFLTs is

$$\Lambda(E_2[200]) = (\lambda p.\lambda f.\lambda x.p\ p\ p\ (\lambda y.p\ p\ (\lambda z.p\ f\ z)\ (p\ p\ f\ y))\ (p\ p\ (\lambda w.p\ f\ w)\ x))\ \mathcal{C}(2) \quad (5.1)$$

The size of the above (5.1) is 48, which is much smaller than $\#\mathcal{C}(200) = 403$.

For TAE E_φ, the following lemma holds.

Lemma 4. *For any TAE E_φ, its CFLT exists.*

Proof. We prove the lemma by induction. Considering λ-terms stated in Corollary 3, for $E_\varphi = \varphi$ and $E_\varphi = {}^t\varphi$, their CFLTs are obtained as

$$\begin{aligned} \Lambda(\varphi) &= (\lambda p.\lambda f.\lambda x.p\ f\ x)\ \mathcal{C}(\varphi) \quad \text{and} \\ \Lambda({}^t\varphi) &= (\lambda p.\lambda f.\lambda x.\overbrace{p\ p\ \cdots\ p}^{t}\ f\ x)\ \mathcal{C}(\varphi), \end{aligned}$$

respectively.

Let $E_\varphi^{(1)}$ and $E_\varphi^{(2)}$ be TAEs. Then, we assume that there are CFLTs of $E_\varphi^{(1)}$ and $E_\varphi^{(2)}$. Moreover, by Definition 13, we assume that $\Lambda(E_\varphi^{(1)}) = (\lambda p.\lambda f.\lambda x.\hat{M}^{(1)})\ \mathcal{C}(\varphi)$ and $\Lambda(E_\varphi^{(2)}) = (\lambda p.\lambda f.\lambda x.\hat{M}^{(2)})\ \mathcal{C}(\varphi)$. Here, CFLTs of $E_\varphi + E_\varphi$ and $E_\varphi \cdot E_\varphi$ are obtained as

$$\begin{aligned} \Lambda(E_\varphi^{(1)} + E_\varphi^{(2)}) &= (\lambda p.\lambda f.\lambda x.(\lambda y.\hat{M}^{(1)}[x := y])\ ((\lambda y.\hat{M}^{(2)}[x := y])\ x))\ \mathcal{C}(\varphi) \quad \text{and} \\ \Lambda(E_\varphi^{(1)} \cdot E_\varphi^{(2)}) &= (\lambda p.\lambda f.\lambda x.(\lambda g.\lambda y.\hat{M}^{(1)}[f := g, x := y])\ (\lambda y.\hat{M}^{(2)}[x := y])\ x)\ \mathcal{C}(\varphi), \end{aligned}$$

respectively. □

For compaction, next we introduce *simplification* of λ-terms.

Definition 14 (Simplification of λ-terms)**.** *Let x be a variable, M_1 and M_2 be λ-terms, and $\xrightarrow{\sim}_\beta$ be a special β-reduction from $(\lambda x.M_1)\ M_2$ to $M_1[x := M_2]$ which defined only if at least one of the following holds:*

1. *x occurs in M_1 only once (in other words, x is linear in M_1),*
2. *$\#M_2 = 1$.*

Then, simplification *is reflexive transitive closure of $\xrightarrow{\sim}_\beta$.*

By using simplification for $(\lambda x.M_1)\ M_2$, we can reduce the size of the λ-term since $\#M_1[x := M_2] = \#M_1 + \#M_2 - 1$ (if x occurs in M_1 only once) or $\#M_1[x := M_2] = \#M_1$ (if $\#M_2 = 1$) holds, while $\#((\lambda x.M_1)\ M_2) = \#M_1 + \#M_2 + 2$.

From the above discussion, we can translate a given TAE E_φ into a compact λ-term by applying simplification after translating E_φ into the CFLT in the manner of the proof of Lemma 4. We show the translating algorithm in Algorithm 2. The time complexity of Algorithm 2 depends on the total number of the occurrences of addition and multiple in the input TAE.

5.3.4 Recursive Tetrational Partitioning (RTP)

We consider how we obtain an effective $E_\varphi[\bar{n}]$ for given $\bar{n}$ and φ in the sense that the CFLT becomes more compact. $E_\varphi[\bar{n}]$ is more effective if it consists of fewer arithmetic operations because the size of the CFLT increases with the number of arithmetic operations occurring in it. For example, for $\bar{n} = 12$ and $\varphi = 3$, $3 \cdot 3 + 3$ is more effective than $3 + 3 + 3 + 3$. We propose *recursive tetrational partitioning (RTP)*, which is a heuristic method to find such an effective TAE for given $\bar{n}$ and φ.

Algorithm 2 Translation from E_φ to its CFLT

Input: E_φ

Output: $\Lambda(E_\varphi)$

function TRANSLATE(E_φ)
  if $E_\varphi = \varphi$ **then**
    return $(\lambda p.\lambda f.\lambda x.p\ f\ x)$
  else if $E_\varphi = {}^t\varphi$ **then**
    return $(\lambda p.\lambda f.\lambda x.\overbrace{p\ p\ \cdots\ p}^{t}\ f\ x)$
  else if $E_\varphi = E_\varphi^{(1)} + E_\varphi^{(2)}$ **then**
    $(\lambda p.\lambda f.\lambda x.\hat{M}^{(1)}) :=$ TRANSLATE($E_\varphi^{(1)}$)
    $(\lambda p.\lambda f.\lambda x.\hat{M}^{(2)}) :=$ TRANSLATE($E_\varphi^{(2)}$)
    return $(\lambda p.\lambda f.\lambda x.(\lambda y.\hat{M}^{(1)}[x := y])\ ((\lambda y.\hat{M}^{(2)}[x := y])\ x))$
  else if $E_\varphi = E_\varphi^{(1)} \cdot E_\varphi^{(2)}$ **then**
    $(\lambda p.\lambda f.\lambda x.\hat{M}^{(1)}) :=$ TRANSLATE($E_\varphi^{(1)}$)
    $(\lambda p.\lambda f.\lambda x.\hat{M}^{(2)}) :=$ TRANSLATE($E_\varphi^{(2)}$)
    return $(\lambda p.\lambda f.\lambda x.(\lambda g.\lambda y.\hat{M}^{(1)}[f := g, x := y])\ (\lambda y.\hat{M}^{(2)}[x := y])\ x)$
  end if
end function

$M :=$ TRANSLATE(E_φ)
$M^* :=$ the λ-term generated by simplification of M
return $(M^*\ \mathcal{C}(\varphi))$

Definition 15 (Recursive tetrational partitioning (RTP))**.** *Let n and $\varphi \geq 2$ be natural numbers, and $r = n \bmod \varphi$. Then, recursive tetrational partitioning (RTP) is a method of decomposing n into $E_\varphi[n-r]+r$. Let $T_\varphi[n]$ denote a TAE generated by RTP, then RTP recursively decompose n in the manner described as follows:*

$$T_\varphi[n] := \begin{cases} n & (\textit{if } \ n \leq \varphi), \\ {}^k\varphi \cdot T_\varphi[p_k - r_k] + (\overbrace{{}^k\varphi + \cdots + {}^k\varphi}^{r_k}) & \\ \qquad + \cdots + {}^1\varphi \cdot T_\varphi[p_1 - r_1] + (\overbrace{{}^1\varphi + \cdots + {}^1\varphi}^{r_1}) + r & (\textit{otherwise}), \end{cases}$$

where k is the maximum natural number such that ${}^k\varphi \leq n$, and for $1 \leq i \leq k$, p_i is the natural number such that $0 \leq p_i \cdot {}^i\varphi < {}^{i+1}\varphi$ and $r_i = p_i \bmod \varphi$. Here, if $p_i - r_i = 0$ or 1, we omit the corresponding terms or coefficients in the generated TAE, respectively.

For example, for $n = 201$ and $\varphi = 2$, $T_\varphi[n] = T_2[201] = {}^32 \cdot ({}^22 \cdot 2 + {}^22) + {}^22 \cdot 2 + 1$. In Definition 15, note that each coefficient $(p_i - r_i)$ of ${}^i\varphi$ is a multiple of φ, and then, each $T_\varphi[p_i - r_i]$ generates just $E_\varphi[p_i - r_i]$ without any remainder.

Theorem 10. *For given natural numbers n and $\varphi \geq 2$, $T_\varphi[n]$ is uniquely determined.*

Proof. We prove the statement by induction. For $n \leq \varphi$, the statement clearly holds. Otherwise, let k be the maximum natural number such that ${}^k\varphi \leq n$, $\bar{n} = n - (n \bmod \varphi)$, and $P = \{p_k, p_{k-1}, \cdots, p_1\}$ be a set such that

$$\sum_{i=1}^{k} p_i \cdot {}^i\varphi = \bar{n}. \tag{5.2}$$

Then, it is sufficient to prove that P is uniquely determined for $\bar{n}$ since k, $\bar{n}$, and each r_i are uniquely determined from n, φ, and P. By Definition 15, $0 \leq p_i \cdot {}^i\varphi < {}^{i+1}\varphi$ holds for $1 \leq i \leq k$. We assume that p_k is not the maximum natural number such that

$p_k \cdot {}^k\varphi \leq \bar{n}$, then

$$\bar{n} - p_k \cdot {}^k\varphi \geq {}^k\varphi + (\bar{n} \bmod {}^k\varphi) \tag{5.3}$$

holds. On the other hand,

$$\sum_{i=1}^{k-1} p_i \cdot {}^i\varphi < \sum_{i=2}^{k} {}^i\varphi \leq {}^k\varphi + (\bar{n} \bmod {}^k\varphi) \tag{5.4}$$

also holds. By (5.3) and (5.4), $\sum_{i=1}^{k} p_i \cdot {}^i\varphi < \bar{n}$ holds. However, this contradicts to (5.2). Therefore, p_k must be the maximum one. Similar discussion holds for $p_{k-1}, p_{k-2}, \cdots, p_2$, that is, for $2 \leq i \leq k-1$, p_i is the maximum natural number such that $p_i \cdot {}^i\varphi \leq \bar{n} \bmod {}^{i+1}\varphi$. By (5.2), p_1 is unique if $p_k, p_{k-1} \cdots, p_2$ are unique. Hence, P is uniquely determined for $\bar{n}$ and k. □

Theorem 11. *For given natural numbers n and $\varphi \geq 2$, there exists a CFLT of $T_\varphi[n]$.*

Proof. By Definition 15, $T_\varphi[n]$ has form of $E_\varphi[\bar{n}]+r$, where $r = n \bmod \varphi$ and $\bar{n} = n-r$. By Lemma 4, for $E_\varphi[\bar{n}]$, there is a CFLT $\Lambda(E_\varphi[\bar{n}]) = (\lambda p.\lambda f.\lambda x.\hat{M})\ \mathcal{C}(\varphi)$. Then, a CFLT of $T_\varphi[n]$ is obtained as

$$\Lambda(T_\varphi[n]) = (\lambda p.\lambda f.\lambda x.\hat{M}[x := (\overbrace{f\ (f\ \cdots\ (f}^{r}\ x)\cdots))])\ \mathcal{C}(\varphi).$$

□

For the size of $\Lambda(T_\varphi[n])$, the following two lemmas hold.

Lemma 5. *For a given $T_\varphi[n]$ with natural numbers n and $\varphi \geq 2$, let N_a and N_m be the numbers of addition and multiplication occurring in $T_\varphi[n]$, respectively. Also, for tetration ${}^{n_2}n_1$, we call n_2 second argument of tetration, then let N_t be the sum of second arguments of all tetration occurring in $T_\varphi[n]$. Then,*

$$\#\Lambda(T_\varphi[n]) \leq 14N_a + 8N_m + 2N_t + 2r + 2\varphi + 6$$

holds with $r = n \bmod \varphi$.

Proof. By Definition 13, we assume that $\Lambda(E_\varphi^{(i)}) = (\lambda p.\lambda f.\lambda x.\hat{M}^{(i)})\ \mathcal{C}(\varphi)$. Then, the following holds:

$$\begin{aligned}
\#\Lambda({}^t\varphi) &= \#((\lambda p.\lambda f.\lambda x.\overbrace{p\ p\ \cdots\ p}^{t}\ f\ x)\ \mathcal{C}(\varphi)) \\
&= 2t + 7 + \#\mathcal{C}(\varphi), && (5.5) \\
\#\Lambda({}^t\varphi \cdot E_\varphi^{(1)}) &= \#((\lambda p.\lambda f.\lambda x.\overbrace{p\ p\ \cdots\ p}^{t}\ (\lambda y.\hat{M}^{(1)}[x := y])\ x)\ \mathcal{C}(\varphi)) && (5.6) \\
&= \#\hat{M}^{(1)} + 2t + 7 + \#\mathcal{C}(\varphi), && (5.7) \\
\#\Lambda(E_\varphi^{(1)} + E_\varphi^{(2)}) &= \#((\lambda p.\lambda f.\lambda x.(\lambda y.\hat{M}^{(1)}[x := y])\ ((\lambda y.\hat{M}^{(2)}[x := y])\ x))\ \mathcal{C}(\varphi)) \\
&= \#\hat{M}^{(1)} + \#\hat{M}^{(2)} + 9 + \#\mathcal{C}(\varphi). && (5.8)
\end{aligned}$$

By Definition 15, $T_\varphi[n]$ has the form of $E_\varphi + r$, where E_φ is TAE inductively defined as $E_\varphi ::= {}^t\varphi \mid {}^t\varphi \cdot E_\varphi \mid E_\varphi + E_\varphi$ with natural number t. Therefore, by Equations (5.5), (5.7), and (5.8),

$$\begin{aligned}
\#\Lambda(E_\varphi) &\leq 6N_a + 4N_m + 2N_t + 3 + 4\sum_i \hat{M}^{(i)} + \#\mathcal{C}(\varphi) \\
&= 6N_a + 4N_m + 2N_t + 3 + 4(2N_a + N_m) + \#\mathcal{C}(\varphi) \\
&= 14N_a + 8N_m + 2N_t + 3 + \#\mathcal{C}(\varphi). && (5.9)
\end{aligned}$$

holds. Note that, in (5.5), (5.7), and (5.8), the constant size 3 arise from $\lambda p.M$, $\lambda f.M$, and functional application of λ-abstraction and Church numeral $\mathcal{C}(\varphi)$, which do not appear in higher level λ-term as $\hat{M}^{(i)}$. Also note that $\overbrace{p\ p\ \cdots\ p}^{t}$ occurring in (5.6) is the λ-term generated by simplification. Then,

$$\begin{aligned}
\#\Lambda(T_\varphi[n]) &\leq 14N_a + 8N_m + 2N_t + 2r + 3 + \#\mathcal{C}(\varphi) \\
&= 14N_a + 8N_m + 2N_t + 2r + 2\varphi + 6
\end{aligned}$$

Table 5.1: n, $\#\mathcal{C}(n)$, and $\#\Lambda(T_\varphi[n])$ for $8 < n \leq 15$ and $\varphi = 3$.

$\boldsymbol{n}$	$\boldsymbol{\#\mathcal{C}(n)}$	$\boldsymbol{\#\Lambda(T_\varphi[n])}$	$\boldsymbol{n}$	$\boldsymbol{\#\mathcal{C}(n)}$	$\boldsymbol{\#\Lambda(T_\varphi[n])}$
9	21	20	13	29	26
10	23	22	14	31	28
11	25	24	15	33	28
12	27	24			

follows (5.9). □

Lemma 6. *For a given natural number $n > 8$, there is a natural number φ such that $2 \leq \varphi \leq \lfloor\sqrt{n}\rfloor$ and $\#\Lambda(T_\varphi[n]) < \#\mathcal{C}(n)$.*

Proof. Let $\varphi = \lfloor\sqrt{n}\rfloor$ and $r = n \bmod \varphi$. By Definition 15, $T_\varphi[n]$ has the form of $\varphi \cdot \varphi + r$. Then, since $r < \varphi$,

$$\begin{aligned}
\#\Lambda(T_\varphi[n]) &= \#\Lambda(\varphi \cdot \varphi + r) \\
&= \#((\lambda p.\lambda f.\lambda x.p\ (p\ f)\ (\overbrace{f\ (f\ \cdots\ (f}^{r}\ x)\cdots)))\ \mathcal{C}(\varphi)) \\
&= 2r + 2\varphi + 14 \\
&< 4\varphi + 14 \\
&\leq 4\sqrt{n} + 14 \qquad (5.10)
\end{aligned}$$

holds. By (5.10), the statement holds for $n > 15$ because $\#\mathcal{C}(n) = 2n + 3$. For $8 < n \leq 15$, the statement holds when $\varphi = 3$. Table 5.1 shows n, $\#\mathcal{C}(n)$, and $\#\Lambda(T_\varphi[n])$ in such cases. □

5.3.5 Further Compaction

Let $\Lambda(T_\varphi[n]) = M\ \mathcal{C}(\varphi)$. Hereinafter, we call M and $\mathcal{C}(\varphi)$ function term and argument term, respectively, similarly to the λ-terms stated in Corollary 3. Moreover, for a given natural number n, φ^* denotes a natural number such that $\varphi^* \geq 2$ and $\#\Lambda(T_{\varphi^*}[n])$ is smaller than any other $\#\Lambda(T_\varphi[n])$, where $\varphi \neq \varphi^*$.

Lemma 6 implies that there is room for more compaction of $\Lambda(T_{\varphi^*}[n])$ if $\varphi^* > 8$, since there is a natural number $\varphi_1^* \geq 2$ such that $\#\Lambda(T_{\varphi_1^*}[\varphi^*]) < \#\mathcal{C}(\varphi^*)$. This holds recursively, that is, we can obtain a compact λ-term $\mathcal{L}(n)$ such that $\mathcal{L}(n) \longrightarrow_\beta^* \mathcal{C}(n)$ and $\#\mathcal{L}(n) \leq \#\Lambda(T_{\varphi^*}[n])$ as

$$\mathcal{L}(n) := \begin{cases} \Lambda(T_{\varphi^*}[n]) & (\text{if } \ \varphi^* \leq 8), \\ M\ \mathcal{L}(\varphi^*) & (\text{if } \ \varphi^* > 8), \end{cases}$$

where M is the function term of $\Lambda(T_{\varphi^*}[n])$. We show an algorithm generating $\mathcal{L}(n)$ for a given n in Algorithm 3. In line 9, we use Algorithm 2 to obtain $\Lambda(E_\varphi)$. Assume that $\mathcal{O}(\alpha)$ is the time complexity of Algorithm 2, then as shown in Algorithm 3, $\mathcal{O}(\alpha\sqrt{n})$ time is required to obtain φ^* once.

For the size of $\mathcal{L}(n)$, the following theorem holds.

Theorem 12. *For a given natural number n if there are φ and t such that ${}^t\varphi = n$, then $\#\mathcal{L}(n) = \mathcal{O}(\mathrm{slog}_\varphi n)$.*

Proof. By Definition 15 and Lemma 6,

$$\begin{aligned} \#\mathcal{L}(n) &\leq \#\Lambda(T_\varphi[n]) \\ &= \#((\lambda p.\lambda f.\lambda x.\overbrace{p\ p\ \cdots\ p}^{t}\ f\ x)\ \mathcal{C}(\varphi)) \\ &\leq \#((\lambda p.\lambda f.\lambda x.\overbrace{p\ p\ \cdots\ p}^{t}\ f\ x)\ \mathcal{C}(8)) \\ &= 2t + 26. \end{aligned}$$

Algorithm 3 Generation of $\mathcal{L}(n)$ for a given natural number n

Input: n

Output: $\mathcal{L}(n)$

1: **function** GENERATE(n)
2: **if** $n \leq 8$ **then**
3: **return** $\mathcal{C}(n)$
4: **else**
5: $\varphi := 2,\ \ \varphi^* := \varphi,\ \ minsize := \infty,\ \ M := NULL$
6: **while** $\varphi \leq \lfloor \sqrt{n} \rfloor$ **do**
7: $r := n \bmod \varphi$
8: $E_\varphi + r := T_\varphi[n]$
9: $((\lambda p.\lambda f.\lambda x.\hat{M})\ \mathcal{C}(\varphi)) := \Lambda(E_\varphi)$
10: $\Lambda(T_\varphi[n]) := ((\lambda p.\lambda f.\lambda x.\hat{M}[x := (\overbrace{f\ (f\ \cdots\ (f}^{r}\ x)\cdots))])\ \mathcal{C}(\varphi))$
11: **if** $\#\Lambda(T_\varphi[n]) < minsize$ **then**
12: $\varphi^* \leftarrow \varphi$
13: $minsize \leftarrow \#\Lambda(T_\varphi[n])$
14: $M \leftarrow$ the function term of $\Lambda(T_\varphi[n])$
15: **end if**
16: $\varphi \leftarrow \varphi + 1$
17: **end while**
18: $M_{arg} :=$ GENERATE(φ^*)
19: **return** $(M\ M_{arg})$
20: **end if**
21: **end function**
22:
23: **return** GENERATE(n)

holds. Since $t = \mathrm{slog}_\varphi n$, the statement holds. □

As seen in the above Theorem 12, by using proposed method, we can obtain $\mathcal{L}(n)$ which is a tetrationally compact expression of $\mathcal{C}(n)$ for some n. This is properly smaller than $\mathcal{B}(n)$ which we introduced in Definition 11. However, Theorem 12 refers to only special cases of n. For general n, the following theorem holds.

Theorem 13. *For a given natural number n, $\#\mathcal{L}(n) = \mathcal{O}((\mathrm{slog}_2 n)^{\frac{\log n}{\log\log n}})$.*

We need the following lemma to prove Theorem 13.

Lemma 7. *For a given natural number n, let M_{φ^*} and M_2 be function terms of $\Lambda(T_{\varphi^*}[n])$ and $\Lambda(T_2[n])$, respectively. Then, $\#M_{\varphi^*} \leq \#M_2$ holds.*

Proof. According to the definition of sizes of λ-terms, $\#\Lambda(T_{\varphi^*}[n]) = \#M_{\varphi^*} + \#\mathcal{C}(\varphi^*) + 1$ and $\#\Lambda(T_2[n]) = \#M_2 + \#\mathcal{C}(2) + 1$ holds. Therefore, by definition of φ^*, the following holds:

$$\begin{aligned}
\#\Lambda(T_{\varphi^*}[n]) \leq \#\Lambda(T_2[n]) &\iff \#M_{\varphi^*} + \#\mathcal{C}(\varphi^*) + 1 \leq \#M_2 + \#\mathcal{C}(2) + 1 \\
&\iff \#M_{\varphi^*} - \#M_2 \leq \#\mathcal{C}(2) - \#\mathcal{C}(\varphi^*) \\
&\iff \#M_{\varphi^*} - \#M_2 \leq 0.
\end{aligned}$$

The statement follows the above. Note that, according to Definition 15, $T_\varphi[n]$ is only defined for $\varphi \geq 2$. □

Now we can prove Theorem 13.

Proof of Theorem 13. According to the definition of $\mathcal{L}(n)$, we assume that $\mathcal{L}(n) = M_{\varphi^*}\mathcal{L}(\varphi^*)$, where M_{φ^*} is the function term of $\Lambda(T_{\varphi^*}[n])$. According to the definition of sizes of λ-terms, the following holds:

$$\#\mathcal{L}(n) = \#M_{\varphi^*} + \#\mathcal{L}(\varphi^*) + 1. \tag{5.11}$$

First, we consider the size of M_{φ^*}. For a given natural number n, we inductively define a TAE $\hat{T}_2(n)$ as follows:

$$\hat{T}_2(n) := \begin{cases} n & (\text{if } \ n \leq 2), \\ {}^k2 \cdot \hat{T}_2(p_k - r_k) + {}^k2 + \cdots + {}^12 \cdot \hat{T}_2(p_1 - r_1) + {}^12 + 2 & (\text{otherwise}). \end{cases}$$

where k is the maximum natural number such that ${}^k2 \leq n$, and for $1 \leq i \leq k$, each p_i is the natural number such that $0 \leq p_i \cdot {}^i2 < {}^{i+1}2$ and $r_i = p_i \bmod 2$. If $p_i = 0$ or 1, we omit the corresponding terms or coefficients in the TAE, similarly to $T_2[n]$. While $\hat{T}_2(n)$ is not $E_2[n]$ (since there is a case that the evaluated result of $\hat{T}_2(n)$ is not equal to n), $\hat{T}_2(n)$ is clearly a TAE E_φ. Therefore, by Lemma 4, a CFLT $\Lambda(\hat{T}_2(n))$ exists. By Lemma 5, the upper bound of $\#\Lambda(\hat{T}_2(n))$ depends on the number of the arithmetic operations occurring in $\hat{T}_2(n)$. Let N_a and N_m be the numbers of addition and multiplication occurring in $T_2[n]$, respectively. Also, for tetration ${}^{n_2}n_1$, we call n_2 second argument of tetration, then let N_t be the sum of second arguments of all tetration occurring in $T_2[n]$. Let us denote $\hat{N}_a$, $\hat{N}_m$, and $\hat{N}_t$ similarly for $\hat{T}_2(n)$. Here, $N_a \leq \hat{N}_a$, $N_m \leq \hat{N}_m$, and $N_t \leq \hat{N}_t$ hold, since in $T_2[n]$, no terms increase compared to $\hat{T}_2(n)$ while the terms ${}^i2$ following ${}^i2 \cdot \hat{T}_2(n)$ is reduced if $r_i = 0$. Then,

$$\begin{aligned} \#\Lambda(T_2[n]) &\leq \max(\#\Lambda(\hat{T}_2(n))) \\ &\leq 14\hat{N}_a + 8\hat{N}_m + 2\hat{N}_t + 11. \end{aligned}$$

follows Lemma 5. Note that $(n \bmod 2) \leq 1$. Let $\hat{N}(n) = 14\hat{N}_a + 8\hat{N}_m + 2\hat{N}_t + 11$, then by the above,

$$\begin{aligned} \#\Lambda(T_2[n]) &\leq \sum_{i=1}^{k} (14 \cdot 2 + 8 \cdot 1 + 2 \cdot 2i + \hat{N}(p_i - r_i) + 11) \\ &\leq 2k^2 + 13k + k\hat{N}(p_k - r_k). \end{aligned} \tag{5.12}$$

holds. We assume that ρ is the depth of the recursion in $\hat{T}_2(n)$. Then, $k\hat{N}(p_k - r_k) = \mathcal{O}(k^\rho)$ holds.

Therefore, by (5.12),

$$\#\Lambda(T_2[n]) = \mathcal{O}(k^\rho). \tag{5.13}$$

holds. Here, $({}^k2)^\rho \leq n \iff \rho \log {}^k2 \leq \log n$ holds. Thus, by Corollary 2,

$$\rho \leq \frac{\log n}{\log {}^k2} < \frac{\log n}{\log {}^{(\mathrm{slog}_2 n)-1}2} = \frac{\log n}{\log \log n}$$

holds. According to Definition 6, $k \leq \mathrm{slog}_2 n$. Then,

$$\#\Lambda(T_2[n]) = \mathcal{O}((\mathrm{slog}_2 n)^{\frac{\log n}{\log \log n}}) \tag{5.14}$$

follows (5.13). By Lemma 7, $\#M_{\varphi^*}$ is bounded by $\Lambda(T_2[n])$ since the function term of $\Lambda(T_2[n])$ is smaller than $\Lambda(T_2[n])$ itself. Then, by (5.14), we obtain

$$\#M_{\varphi^*} = \mathcal{O}((\mathrm{slog}_2 n)^{\frac{\log n}{\log \log n}}). \tag{5.15}$$

Second, we consider the size of $\mathcal{L}(\varphi^*)$. If $\varphi^* \leq 8$, it is constant since $\mathcal{L}(\varphi^*) = \mathcal{C}(\varphi^*)$ and $\#\mathcal{C}(\varphi^*) \leq \mathcal{C}(8) = 19$. Otherwise, $\#\mathcal{L}(\varphi^*) \leq \Lambda(T_{\varphi_1^*}[\varphi^*])$ holds with a natural number φ_1^*. By the definition of $\mathcal{L}(n)$, it recursively holds until $\varphi_i^* \leq 8$. Thus, by (5.11) and (5.15),

$$\begin{aligned} \#\mathcal{L}(\varphi^*) \;=\; & \mathcal{O}((\mathrm{slog}_2\varphi^*)^{\frac{\log \varphi^*}{\log \log \varphi^*}}) + \mathcal{O}((\mathrm{slog}_2\varphi_1^*)^{\frac{\log \varphi_1^*}{\log \log \varphi_1^*}}) \\ & + \cdots + \mathcal{O}((\mathrm{slog}_2\varphi_m^*)^{\frac{\log \varphi_m^*}{\log \log \varphi_m^*}}) + m + C \end{aligned} \tag{5.16}$$

holds, where C is a constant and m is a natural number such that $\varphi_m^* \leq 8$. Here, by Lemma 5, $\varphi^* \leq \lfloor\sqrt{n}\rfloor, \varphi_1^* \leq \lfloor\sqrt{\varphi^*}\rfloor, \varphi_2^* \leq \lfloor\sqrt{\varphi_1^*}\rfloor, \cdots, \varphi_m^* \leq \lfloor\sqrt{{\varphi^*}_{m-1}}\rfloor$ holds, then the recursion depth m is $\mathcal{O}(\log n)$. Hence,

$$\#\mathcal{L}(\varphi^*) = \mathcal{O}((\mathrm{slog}_2\varphi^*)^{\frac{\log \varphi^*}{\log \log \varphi^*}}) + \mathcal{O}(\log n) + C \tag{5.17}$$

follows (5.16). Note that $\sum_{i=1}^{\infty} F(x_i) < 2F(x_1)$ holds for function $F(x)$ and natural number x such that $F(x_i) \geq 0$ and $x_{i+1} \leq \sqrt{x_i}$ for $1 \leq i$.

As a result, by (5.12), (5.15), and (5.17), we obtain

$$\begin{aligned} \#\mathcal{L}(n) &= \mathcal{O}((\mathrm{slog}_2 n)^{\frac{\log n}{\log\log n}}) + \mathcal{O}((\mathrm{slog}_2 \varphi^*)^{\frac{\log \varphi^*}{\log\log \varphi^*}}) + \mathcal{O}(\log n) + C \\ &= \mathcal{O}((\mathrm{slog}_2 n)^{\frac{\log n}{\log\log n}}) \end{aligned}$$

and the statement holds. □

5.4 Application to Higher-Order Compression and Comparative Experiments

We implemented our method stated in Section 5.3, and conducted an experiment compared to the binary expression on λ-terms stated in Definition 11 and λ-terms generated by using the method stated in [104] (we call the method YKS).

Let $\mathcal{B}(n)$ denote the binary expression of n on λ-terms. As seen in Section 5.3, our proposed method generates a compact λ-term $\mathcal{L}(n)$ for a given $\mathcal{C}(n)$. While $\mathcal{C}(n)$ has the size of $\Theta(n)$, the size of the λ-term $\mathcal{L}(n)$ achieves $\mathcal{O}(\mathrm{slog}_2 n)$ for some n, as shown in Theorem 12. However, as shown in Theorem 13, the size of $\mathcal{L}(n)$ becomes $\mathcal{O}((\mathrm{slog}_2 n)^{\log n/\log\log n})$ for general n, which is greater than that of $\mathcal{B}(n)$, $\Theta(\log n)$. On the other hand, YKS is a method of higher-order compression, which generates an extended λ-term for a given text. Here, a *text* is an element of Σ^*, where Σ is an *alphabet*. Let $L_{\mathrm{YKS}}(n)$ be the generated extended λ-term by using YKS for a given text $\mathtt{a}^n\mathtt{c}$ with $\mathtt{a}, \mathtt{c} \in \Sigma$. Then, we can find that the size of $L_{\mathrm{YKS}}(n)$ achieves $\mathcal{O}(\mathrm{slog}_2 n)$ for some n, similarly to $\mathcal{L}(n)$.

We define *extended λ-terms* following to [104]. We show that the application of $\mathcal{B}(n)$ and $\mathcal{L}(n)$ to extended λ-terms is directly possible.

Definition 16 (Extended λ-terms and their sizes)**.** *Let x be a variable and $a \in \Sigma$. Then, extended λ-term $\tilde{M}$ is defined inductively as follows:*

$$\tilde{M} ::= a \mid x \mid \lambda x.\tilde{M} \mid \tilde{M}_1\ \tilde{M}_2.$$

The size of $\tilde{M}$, denoted by $\#\tilde{M}$, is inductively defined as follows:

$$\#a = 1, \quad \#x = 1, \quad \#(\lambda x.\tilde{M}) = \#\tilde{M} + 1, \quad \#(\tilde{M}_1\ \tilde{M}_2) = \#\tilde{M}_1 + \#\tilde{M}_2 + 1.$$

Similar to normal λ-terms, we define β-reduction, variable replacement notation $\tilde{M}_1[x := \tilde{M}_2]$, and simplification on extended λ-terms. Clearly, we can regard any normal λ-terms as extended λ-terms.

In higher-order compression, a given text $\mathtt{a}^n\mathtt{c}$ is regarded as an extended λ-term $\tilde{S}(n) = (\overbrace{\mathtt{a}\ (\mathtt{a}\ \cdots\ (\mathtt{a}}^{n}\ \mathtt{c})\cdots))$, then we generate a compact extended λ-term $\tilde{M}$ such that $\tilde{M} \longrightarrow_\beta^* \tilde{S}(n)$. From $\mathcal{B}(n)$ and $\mathcal{L}(n)$, we can obtain extended λ-terms $\tilde{\mathcal{B}}(n)$ and $\tilde{\mathcal{L}}(n)$ such that $\tilde{\mathcal{B}}(n) \longrightarrow_\beta^* \tilde{S}(n)$ and $\tilde{\mathcal{L}}(n) \longrightarrow_\beta^* \tilde{S}(n)$, respectively, as follows. By Definition 11 and the definition of $\mathcal{L}(n)$, we assume that $\mathcal{B}(n) = (\lambda p.\lambda f.\lambda x.\hat{M}_{\mathcal{B}})\ M'_{\mathcal{B}}$ and $\mathcal{L}(n) = (\lambda p.\lambda f.\lambda x.\hat{M}_{\mathcal{L}})\ M'_{\mathcal{L}}$. Then, $\tilde{\mathcal{B}}(n)$ and $\tilde{\mathcal{L}}(n)$ are obtained as

$$\tilde{\mathcal{B}}(n) = (\lambda p.\hat{M}_{\mathcal{B}}[f := \mathtt{a}, x := \mathtt{c}])\ M'_{\mathcal{B}} \quad \text{and} \quad \tilde{\mathcal{L}}(n) = (\lambda p.\hat{M}_{\mathcal{L}}[f := \mathtt{a}, x := \mathtt{c}])\ M'_{\mathcal{L}}\ ,$$

respectively. Note that $\mathcal{B}(n)\ \mathtt{a}\ \mathtt{c} \longrightarrow_\beta^* \tilde{S}(n)$ and $\mathcal{L}(n)\ \mathtt{a}\ \mathtt{c} \longrightarrow_\beta^* \tilde{S}(n)$ holds, since $\mathcal{B}(n) \longrightarrow_\beta^* \mathcal{C}(n)$, $\mathcal{L}(n) \longrightarrow_\beta^* \mathcal{C}(n)$, and $\mathcal{C}(n)\ \mathtt{a}\ \mathtt{c} \longrightarrow_\beta^* \tilde{S}(n)$. Here, $\tilde{\mathcal{B}}(n)$ and $\tilde{\mathcal{L}}(n)$ are the results of simplification for $\mathcal{B}(n)\ \mathtt{a}\ \mathtt{c}$ and $\mathcal{L}(n)\ \mathtt{a}\ \mathtt{c}$, respectively.

Figures 5.1 and 5.2 show the experimental results. In both of them, "Binary", "YKS", and "Proposed" denote the methods using the binary expression on λ-terms,

YKS, and ours, respectively. The horizontal axis shows the repetition number n of given text $\mathtt{a}^n\mathtt{c}$. Then, we compare the sizes of the generated extended λ-terms by using these three methods, for the given $\mathtt{a}^n\mathtt{c}$.

In Figure 5.1, the vertical axis shows the sizes of $\tilde{\mathcal{B}}(n)$, $L_{\mathrm{YKS}}(n)$, and $\tilde{\mathcal{L}}(n)$. The inequality $\#\tilde{\mathcal{L}}(n) \leq \#\tilde{\mathcal{B}}(n)$ holds in 5187 out of 10,000 cases within the range $[1, 10000]$. The average ratio $\#\tilde{\mathcal{L}}(n)/\#\tilde{\mathcal{B}}(n)$ for the range is approximately 0.9962. Similarly, the inequality $\#\tilde{\mathcal{L}}(n) \leq \#L_{\mathrm{YKS}}(n)$ holds in 5959 cases, and the average ratio $\#\tilde{\mathcal{L}}(n)/\#L_{\mathrm{YKS}}(n)$ is approximately 0.9321.

In Figure 5.2, the vertical axis shows the ratio (the average size of cumulative sum of extended λ-terms from 1 to n)$/\#\tilde{\mathcal{B}}(n)$. That is, Figure 5.2 shows how $\#L_{\mathrm{YKS}}(n)$ and $\#\tilde{\mathcal{L}}(n)$ increase compared to $\#\tilde{\mathcal{B}}(n)$ on average. As can be seen, $\#\tilde{\mathcal{L}}(n)$ tends to be greater than $\#\tilde{\mathcal{B}}(n)$ when n is greater than $10,000$. We consider that it is consistent with the theoretical upper bound $\mathcal{O}((\mathrm{slog}_2 n)^{\log n/\log\log n})$, stated in Theorem 13.

5.5 Conclusions

In this chapter, we addressed the problem of compaction of Church numerals. For given natural number n, by using proposed RTP, we decompose n into an arithmetic expression, which enables to obtain a compact λ-term leading to the Church numeral of n. We proved that the size of the obtained λ-term becomes $\mathcal{O}((\mathrm{slog}_2 n)^{\log n/\log\log n})$. Moreover, we experimentally confirmed that the λ-terms produced by our method tend to be smaller than binary expressions on λ-terms on average, when the given number is less than approximately $10,000$.

The compaction of Church numerals can be applied to higher-order compression, that uses extended λ-terms as the data model. In the procedure of higher-order com-

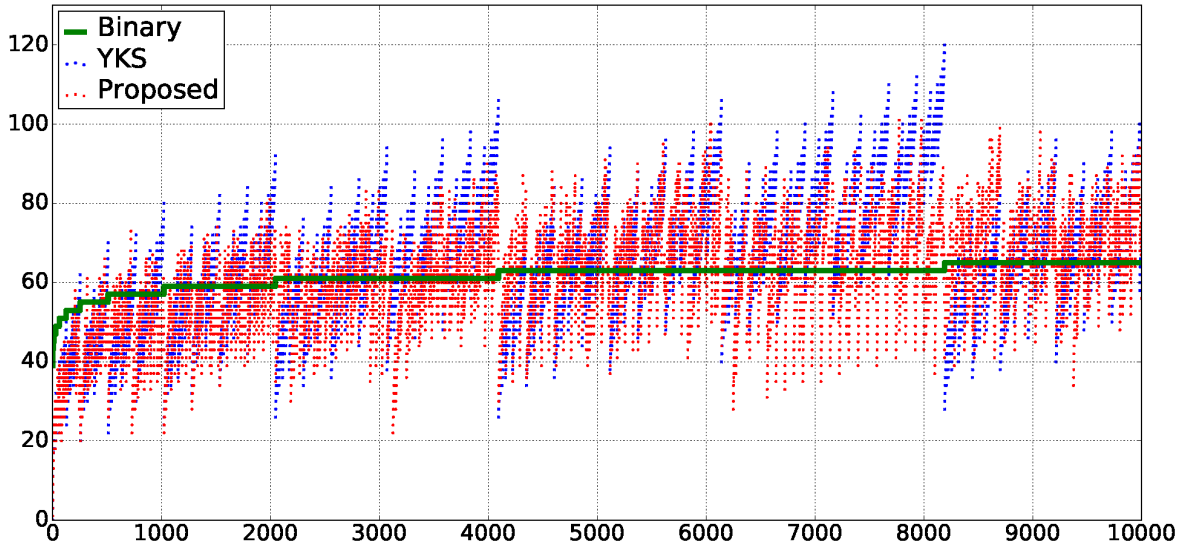

Figure 5.1: Term size for integer n.

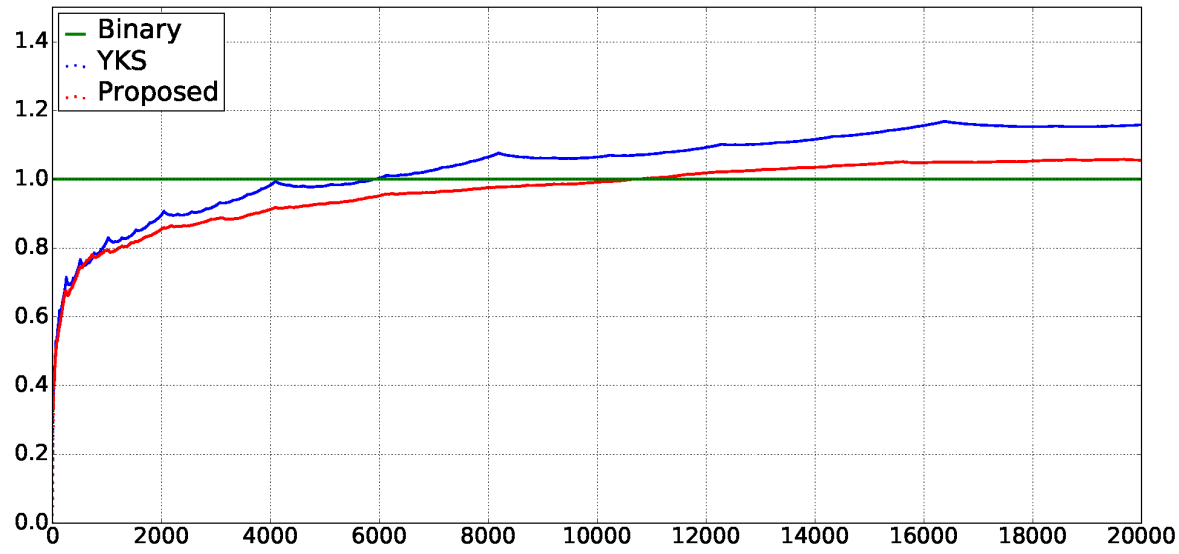

Figure 5.2: Average ratio to binary expression.

pression, a repetitive part in the input can be represented as a λ-term with $\mathcal{C}(n)$. Thus, efficient compaction of $\mathcal{C}(n)$ will help for improving the compression performance of higher-order compression.

Chapter 6

Conclusions

6.1 Summary

This thesis studied lossless compression methods for repetitive data, namely, repetition-aware lossless compression techniques. We discussed on three grammar compression frameworks whose differences are the formal grammars used as the description of the compressed data. We considered a context-free grammar (CFG), a run-length context-free grammar (RLCFG), and a functional program described by λ-term in Chapter 3, Chapter 4, and Chapter 5, respectively.

In Chapter 3, we approached to the problem of repetition-aware compression on CFG-based grammar compression. We analyzed a famous algorithm, RePair, and on the basis of the analysis, we designed a novel variant of RePair, called MR-RePair. We implemented MR-RePair and experimentally confirmed the effectiveness of MR-RePair especially for highly repetitive texts.

In Chapter 4, we addressed further improvement of compression performance via the framework of RLCFG-based grammar compression. In the chapter, we designed a com-

pression algorithm using RLCFG, called RL-MR-RePair. Furthermore, we proposed an encoding scheme for MR-RePair and RL-MR-RePair. The experimental results demonstrated the high compression performance of RL-MR-RePair and the proposed encoding scheme.

In Chapter 5, we studied on the framework of higher-order compression, which is a grammar compression using a λ-term as the formal grammar. We presented a method to obtain a compact λ-term representing a natural number. Obtaining a compact representation of natural numbers can improve the compression effectiveness of repetition, the most fundamental repetitive structure. For given natural number n, we proved that the size of the obtained λ-term becomes $\mathcal{O}(\mathrm{slog}_2 n)$ in the best case and $\mathcal{O}(\mathrm{slog}_2 n)^{\log n/\log\log n}$ in the worst case.

6.2 Towards the Future

We considered the grammar-based lossless compression problem through using formal grammars other than a CFG, while a CFG has been mainly used as the dictionary model. Related to the approach, reconstruction of the hierarchy of formal grammar for lossless compression may be a most challenging task. The study will include the establishment of a new formal grammar for efficient grammar compression. Also, other finer but important future studies related to this work are listed below.

In Chapter 3, we analyzed RePair and defined the greatest size difference of any two possible grammars that can be generated by RePair for a given text, naming it GSDRP. We demonstrated that a lower bound of GSDRP is $\frac{1}{6}(\sqrt{6n+1}+13)$ for a given text of length n. We left improving the lower bound and showing an upper bound of GSDRP as future work.

In Chapter 3 and 4, we developed RePair variants, MR-RePair and RL-MR-RePair. As stated in Section 3.1 and 4.1, RePair practically achieves a higher compression ratio than other existing grammar compression methods. However, like MR-RePair and RL-MR-RePair, it requires a large space for execution. The working space of RePair was recently reduced in [20, 83]. Our future study will explore the development of space-efficient MR-RePair/RL-MR-RePair algorithms.

In Chapter 5, we discussed the size of the obtained λ-term by the proposed method. However, we have not proved the lower bound of the size in the worst case; it remains future work. Furthermore, in data compression, data models are finally encoded in bit sequences. For bit encoding of λ-terms, Tromp [100] proposed a method for untyped λ-terms. Recently, Takeda et al. [99] proposed an efficient encoding scheme for simply-typed λ-terms. Finding an efficient bit encoding for our method is another interesting challenge.

Bibliography

[1] *Proceedings of the IEEE*, 88(11), November 2000.

[2] *IEEE Transactions on Information Theory*, 46(3), May 2000.

[3] *String Processing and Information Retrieval – 15th International Symposium, SPIRE 2008, Melbourne, Australia, November 10–12, 2008. Proceedings*, volume 5280 of *Lecture Notes in Computer Science (LNCS)*. Springer, Berlin, Heidelberg, November 2008.

[4] *String Processing and Information Retrieval – 19th International Symposium, SPIRE 2012, Cartagena de Indias, Colombia, October 21–25, 2012. Proceedings*, volume 7608 of *Lecture Notes in Computer Science (LNCS)*. Springer, Berlin, Heidelberg, October 2012.

[5] *2014 Data Compression Conference, DCC 2014, Snowbird, UT, USA, March 26–28, 2014*. IEEE, March 2014.

[6] *2017 Data Compression Conference, DCC 2017, Snowbird, UT, USA, April 4–7, 2017*. IEEE, April 2017.

[7] *String Processing and Information Retrieval – 24th International Symposium, SPIRE 2017, Palermo, Italy, September 26–29, 2017. Proceedings*, volume 10508 of *Lecture Notes in Computer Science (LNCS)*. Springer, Cham, September 2017.

[8] *2019 Data Compression Conference, DCC 2019, Snowbird, UT, USA, March 26–29, 2019*. IEEE, March 2019.

[9] Alfred V. Aho, John E. Hopcroft, and Jeffrey Ullman. *Data Structures and Algorithms*. Addison-Wesley Longman Publishing Co., Inc., 1st edition, January 1983.

[10] Alberto Apostolico and Stefano Lonardi. Off-line compression by greedy textual substitution. In *Proceedings of the IEEE* [1], pages 1733–1744.

[11] Djamal Belazzougui and Fabio Cunial. Fast label extraction in the CDAWG. In *String Processing and Information Retrieval – 24th International Symposium, SPIRE 2017, Palermo, Italy, September 26–29, 2017. Proceedings* [7], pages 161–175.

[12] Djamal Belazzougui and Fabio Cunial. Representing the suffix tree with the CDAWG. In *28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*, volume 78 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:13. Schloss Dagstuhl–Leibniz–Zentrum fuer Informatik, July 2017.

[13] Djamal Belazzougui, Fabio Cunial, Travis Gagie, Nicola Prezza, and Mathieu Raffinot. Composite repetition-aware data structures. In *Combinatorial Pattern Matching – 26th Annual Symposium, CPM 2015, Ischia Island, Italy, June 29–July 1, 2015. Proceedings*, volume 9133 of *Lecture Notes in Computer Science (LNCS)*, pages 26–39. Springer, Cham, June 2015.

[14] Tim Bell, Matt Powell, Joffre Horlor, and Ross Arnold. The canterbury corpus/The Large Courpus. `http://corpus.canterbury.ac.nz/descriptions/\#large`, Accessed: October 26, 2018.

[15] Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text Compression*. Prentice-Hall, Inc., 1st edition, January 1990.

[16] Philip Bille, Patrick Hagge Cording, and Inge Li Gørtz. Compact q-gram profiling of compressed strings. *Theoretical Computer Science*, 550:51–58, September 2014.

[17] Philip Bille, Patrick Hagge Cording, and Inge Li Gørtz. Compressed subsequence matching and packed tree coloring. *Algorithmica*, 77(2):336–348, February 2017.

[18] Philip Bille, Travis Gagie, Inge Li Gørtz, and Nicola Prezza. A separation between RLSLPs and LZ77. *Journal of Discrete Algorithms*, 50:36–39, May 2018.

[19] Philip Bille, Inge Li Gørtz, and Nicola Prezza. Practical and effective Re-Pair compression. *CoRR*, abs/1704.08558, April 2017.

[20] Philip Bille, Inge Li Gørtz, and Nicola Prezza. Space-efficient Re-Pair compression. In *2017 Data Compression Conference, DCC 2017, Snowbird, UT, USA, April 4–7, 2017* [6], pages 171–180.

[21] Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random access to grammar-compressed strings and trees. *SIAM Journal of Computing*, 44(3):513–539, May 2015.

[22] Michael Burrows and David J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, May 1994.

[23] Gregory J. Chaitin. On the length of programs for computing finite binary sequences: Statistical considerations. *Journal of the ACM*, 16(1):145–159, January 1969.

[24] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and abhi shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(5):2554–2576, May 2005.

[25] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, September 1956.

[26] Francisco Claude and Gonzalo Navarro. Fast and compact web graph representations. *ACM Transactions on the Web*, 4(4):16:1–16:31, September 2010.

[27] Francisco Claude and Gonzalo Navarro. Self-indexed grammar-based compression. *Fundamenta Informaticae*, 111(3):313–337, January 2011.

[28] Francisco Claude and Gonzalo Navarro. Improved grammar-based compressed indexes. In *String Processing and Information Retrieval – 19th International Symposium, SPIRE 2012, Cartagena de Indias, Colombia, October 21–25, 2012. Proceedings* [4], pages 180–192.

[29] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, July 2009.

[30] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. Wiley-Interscience, 2nd edition, July 2006.

[31] Maxime Crochemore and Wojciech Rytter. *Jewels of Stringology – Text Algorithms*. World Scientific Publishing Co. Pte. Ltd., 1st edition, September 2002.

[32] EMBL – European Bioinformatics Institute (EBI). 1000 Genomes – A deep catalog of human genetic variation. `https://www.internationalgenome.org/`, Accessed: June 13, 2020.

[33] Peter Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, March 1975.

[34] Paolo Ferragina and Gonzalo Navarro. Pizza&Chili Corpus – compressed indexes and their testbeds. `http://pizzachili.dcc.uchile.cl`, Accessed: July 18, 2019.

[35] Travis Gagie, Tomohiro I, Giovanni Manzini, Gonzalo Navarro, Hiroshi Sakamoto, and Yoshimasa Takabatake. Rpair: Rescaling RePair with Rsync. In *String Processing and Information Retrieval – 26th International Symposium, SPIRE 2019, Segovia, Spain, October 7–9, 2019. Proceedings*, volume 11811 of *Lecture Notes in Computer Science (LNCS)*, pages 35–44. Springer, Cham, October 2019.

[36] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in bwt-runs bounded space. *Journal of the ACM*, 67(1):2:1–2:54, January 2020.

[37] Michał Gańczorz. Michał Gańczorz/RepairImproved. `https://bitbucket.org/IguanaBen/repairimproved`, Accessed: October 26, 2018.

[38] Michał Gańczorz and Artur Jeż. Improvements on Re-Pair grammar compressor. In *2017 Data Compression Conference, DCC 2017, Snowbird, UT, USA, April 4–7, 2017* [6], pages 181–190.

[39] Paweł Gawrychowski. Faster algorithm for computing the edit distance between SLP-compressed strings. In *String Processing and Information Retrieval – 19th International Symposium, SPIRE 2012, Cartagena de Indias, Colombia, October 21–25, 2012. Proceedings* [4], pages 229–236.

[40] Rodrigo González and Gonzalo Navarro. Compressed text indexes with fast locate. In *Combinatorial Pattern Matching – 18th Annual Symposium, CPM 2007, London, Canada, July 9–11, 2007. Proceedings*, volume 4580 of *Lecture Notes in Computer Science (LNCS)*, pages 216–227. Springer, Berlin, Heidelberg, July 2007.

[41] Keisuke Goto, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Fast q-gram mining on SLP compressed strings. *Journal of Discrete Algorithms*, 18:89–99, January 2013.

[42] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, September 1952.

[43] Tomohiro I, Takaaki Nishimoto, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Compressed automata for dictionary matching. *Theoretical Computer Science*, 578:30–41, May 2015.

[44] Shunsuke Inenaga, Takashi Funamoto, Masayuki Takeda, and Ayumi Shinohara. Linear-time off-line text compression by longest-first substitution. In *String Processing and Information Retrieval – 10th International Symposium, SPIRE 2003,*

Manaus, Brazil, October 8–10, 2003. Proceedings, volume 2857 of *Lecture Notes in Computer Science (LNCS)*, pages 137–152. Springer, Berlin, Heidelberg, October 2003.

[45] Artur Jeż. Approximation of grammar-based compression via recompression. *Theoretical Computer Science*, 592:115–134, August 2015.

[46] Artur Jeż. Faster fully compressed pattern matching by recompression. *ACM Transactions on Algorithms*, 11(3):20:1–20:43, January 2015.

[47] Ming-Yang Kao, editor. *Encyclopedia of Algorithms*. Springer-Verlag New York, 2nd edition, March 2016.

[48] Dominik Kempa and Nicola Prezza. At the roots of dictionary compression: string attractors. In *STOC 2018: Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, pages 827–840. Association for Computing Machinery, June 2018.

[49] Takuya Kida, Tetsuya Matsumoto, Yusuke Shibata, Masayuki Takeda, Ayumi Shinohara, and Setsuo Arikawa. Collage system: a unifying framework for compressed pattern matching. *Theoretical Computer Science*, 298(1):253–272, April 2003.

[50] John C. Kieffer and En hui Yang. Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform – part one: Without context models. In *IEEE Transactions on Information Theory* [2], pages 755–777.

[51] John C. Kieffer and En hui Yang. Grammar-based codes: a new class of universal lossless source codes. In *IEEE Transactions on Information Theory* [2], pages 737–754.

[52] John C. Kieffer, En hui Yang, Gregory J. Nelson, and Pamela Cosman. Universal lossless compression via multilevel pattern matching. *IEEE Transactions on Information Theory*, 46(5):1227–1245, July 2000.

[53] Naoki Kobayashi, Kazutaka Matsuda, Ayumi Shinohara, and Kazuya Yaguchi. Functional programs as compressed data. *Higher-Order and Symbolic Computation*, 25(1):39–84, March 2012.

[54] Andrei N. Kolmogorov. On tables of random numbers. *Sankhyā, Series A*, 25(4):369–376, December 1963.

[55] Andrei N. Kolmogorov. On tables of random numbers (reprinted from "Sankhya: The Indian Journal of Statistics", Series A, Vol. 25 Part 4, 1963). *Theoretical Computer Science*, 207(2):387–395, November 1998.

[56] N. Jesper Larsson and Alistair Moffat. Off-line dictionary-based compression. In *Proceedings of the IEEE* [1], pages 1722–1732.

[57] Markus Lohrey. Algorithmics on SLP-compressed strings: A survey. *Groups Complexity Cryptology*, 4(2):241–299, December 2012.

[58] Markus Lohrey, Sebastian Maneth, and Roy Mennicke. XML tree structure compression using RePair. *Information Systems*, 38(8):1150–1167, November 2013.

[59] Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, March 2005.

[60] Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, April 2010.

[61] Shirou Maruyama. re-pair – a grammar-based compressor by most-frequent-first substitution. `https://code.google.com/archive/p/re-pair/`, Accessed: October 26, 2018.

[62] Shirou Maruyama, Yasuo Tabei, Hiroshi Sakamoto, and Kunihiko Sadakane. Fully-online grammar compression. In *String Processing and Information Retrieval – 20th International Symposium, SPIRE 2013, Jerusalem, Israel, October 7–9, 2013. Proceedings*, volume 8214 of *Lecture Notes in Computer Science (LNCS)*, pages 218–229. Springer, Cham, October 2013.

[63] Shirou Maruyama, Yohei Tanaka, Hiroshi Sakamoto, and Masayuki Takeda. Context-sensitive grammar transform: Compression and pattern matching. In *String Processing and Information Retrieval – 15th International Symposium, SPIRE 2008, Melbourne, Australia, November 10–12, 2008. Proceedings* [3], pages 27–38.

[64] Takuya Masaki and Takuya Kida. Online grammar transformation based on Re-Pair algorithm. In *2016 Data Compression Conference, DCC 2016, Snowbird, UT, USA, March 30–April 1, 2016*, pages 349–358. IEEE, March 2016.

[65] Alistair Moffat and Andrew Turpin. *Compression and Coding Algorithms*. Springer, US, 1st edition, March 2002.

[66] Torben Æ. Mogensen. An investigation of compact and efficient number representations in the pure lambda calculus. In *Perspectives of System Informatics*

– *4th International Andrei Ershov Memorial Conference, PSI 2001, Akademgorodok, Novosibirsk, Russia, July 2–6, 2001. Revised Papers*, volume 2244 of *Lecture Notes in Computer Science (LNCS)*, pages 205–213. Springer, Berlin, Heidelberg, July 2001.

[67] Ryosuke Nakamura, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Simple linear-time off-line text compression by longest-first substitution. In *2007 Data Compression Conference (DCC 2007), 27–29 March 2007, Snowbird, UT, USA*, pages 123–132. IEEE Computer Society, March 2007.

[68] Gonzalo Navarro. Indexing highly repetitive collections. In *Combinatorial Algorithms – 23rd International Workshop, IWOCA 2012, Tamil Nadu, India, July 19–21, 2012, Revised Selected Papers*, volume 7643 of *Lecture Notes in Computer Science (LNCS)*, pages 274–279. Springer, Berlin, Heidelberg, July 2012.

[69] Gonzalo Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, 1st edition, September 2016.

[70] Gonzalo Navarro. Re-Pair compression and decompression (2010). `https://users.dcc.uchile.cl/~gnavarro/software/index.html`, Accessed: July 18, 2019.

[71] Gonzalo Navarro and Luís M. S. Russo. Re-pair achieves high-order entropy. In *2008 Data Compression Conference (DCC 2008), 25–27 March 2008, Snowbird, UT, USA*, page 537. IEEE Computer Society, March 2008.

[72] Mark Nelson and Jean-Loup Gailly. *The Data Compression Book*. MIS:Press, 2nd edition, December 1995.

[73] Craig G. Nevill-Manning. *Inferring Sequential Structure*. PhD thesis, University of Waikato, May 1996.

[74] Craig G. Nevill-Manning and Ian H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, September 1997.

[75] Craig G. Nevill-Manning, Ian H. Witten, and David Maulsby. Compression by induction of hierarchical grammars. In *Proceedings of the IEEE Data Compression Conference, DCC 1994, Snowbird, Utah, USA, March 29–31, 1994*, pages 244–253. IEEE Computer Society, March 1994.

[76] Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Fully dynamic data structure for LCE queries in compressed space. In *41st International Symposium on Mathematical Foundations of Computer Science (MFCS 2016)*, volume 58 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 72:1–72:15. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, August 2016.

[77] Carlos Ochoa and Gonzalo Navarro. RePair and all irreducible grammars are upper bounded by high-order empirical entropy. *IEEE Transactions on Information Theory*, 65(5):3160–3164, May 2019.

[78] Tatsuya Ohno, Keisuke Goto, Yoshimasa Takabatake, Tomohiro I, and Hiroshi Sakamoto. LZ-ABT: A practical algorithm for α-balanced grammar compression. In *Combinatorial Algorithms – 29th International Workshop, IWOCA 2018, Singapore, July 16–19, 2018, Proceedings*, volume 10979 of *Lecture Notes in Computer Science (LNCS)*, pages 323–335. Springer, Cham, July 2018.

[79] Nicola Prezza. rp: a space-efficient compressor based on the Re-Pair grammar. `https://github.com/nicolaprezza/Re-Pair`, Accessed: July 18, 2019.

[80] Molly Przeworski, Richard R. Hudson, and Anna Di Rienzo. Adjusting the focus on human variation. *Trends in Genetics*, 16(7):296–302, July 2000.

[81] Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1-3):211–222, June 2003.

[82] Wojciech Rytter. Grammar compression, LZ-encodings, and string algorithms with implicit input. In *Automata, Languages and Programming – 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings*, volume 3142 of *Lecture Notes in Computer Science (LNCS)*, pages 15–27. Springer, Berlin, Heidelberg, July 2004.

[83] Kensuke Sakai, Tatsuya Ohno, Keisuke Goto, Yoshimasa Takabatake, Tomohiro I, and Hiroshi Sakamoto. RePair in compressed space and time. In *2019 Data Compression Conference, DCC 2019, Snowbird, UT, USA, March 26–29, 2019* [8], pages 518–527.

[84] David Salomon and Glovanni Motta. *Handbook of Data Compression*. Springer, London, 5th edition, November 2009.

[85] Khalid Sayood. *Introduction to Data Compression*. Morgan Kaufmann Publishers Inc., 1st edition, January 1996.

[86] Khalid Sayood. *Introduction to Data Compression*. Morgan Kaufmann Publishers Inc., 4th edition, October 2012.

[87] Kei Sekine, Hirohito Sasakawa, Satoshi Yoshida, and Takuya Kida. Adaptive dictionary sharing method for Re-Pair algorithm. In *2014 Data Compression Conference, DCC 2014, Snowbird, UT, USA, 26–28 March, 2014* [5], page 425.

[88] Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, July 1948.

[89] Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(4):623–656, October 1948.

[90] Jouni Sirén, Niko Välimäki, Veli Mäkinen, and Gonzalo Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *String Processing and Information Retrieval – 15th International Symposium, SPIRE 2008, Melbourne, Australia, November 10–12, 2008. Proceedings* [3], pages 164–175.

[91] Ray J. Solomonoff. A formal theory of inductive inference. Part I. *Information and Control*, 7(1):1–22, March 1964.

[92] Ray J. Solomonoff. A formal theory of inductive inference. Part II. *Information and Control*, 7(2):224–254, June 1964.

[93] James A. Storer. *Data Compression: Methods and Theory*. Computer Science Press, Inc., 1st edition, May 1987.

[94] Yasuo Tabei, Hiroto Saigo, Yoshihiro Yamanishi, and Simon J. Puglisi. Scalable partial least squares regression on grammar-compressed data matrices. In *KDD '16: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1875–1884. Association for Computing Machinery, August 2016.

[95] Yasuo Tabei, Yoshimasa Takabatake, and Hiroshi Sakamoto. A succinct grammar compression. In *Combinatorial Pattern Matching – 24th Annual Symposium, CPM 2013, Bad Herrenalb, Germany, June 17–19, 2013. Proceedings*, volume 7922 of *Lecture Notes in Computer Science (LNCS)*, pages 235–246. Springer, Berlin, Heidelberg, June 2013.

[96] Yoshimasa Takabatake, Tomohiro I, and Hiroshi Sakamoto. A Space-Optimal Grammar Compression. In *25th Annual European Symposium on Algorithms (ESA 2017)*, volume 87 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 67:1–67:15. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, September 2017.

[97] Yoshimasa Takabatake, Yasuo Tabei, and Hiroshi Sakamoto. Online pattern matching for string edit distance with moves. In *String Processing and Information Retrieval – 21st International Symposium, SPIRE 2014, Ouro Preto, Brazil, October 20–22, 2014. Proceedings*, volume 8799 of *Lecture Notes in Computer Science (LNCS)*, pages 203–214. Springer, Cham, October 2014.

[98] Takuya Takagi, Keisuke Goto, Yuta Fujishige, Shunsuke Inenaga, and Hiroki Arimura. Linear-size CDAWG: New repetition-aware indexing and grammar compression. In *String Processing and Information Retrieval – 24th International Symposium, SPIRE 2017, Palermo, Italy, September 26–29, 2017. Proceedings* [7], pages 304–316.

[99] Kotaro Takeda, Naoki Kobayashi, Kazuya Yaguchi, and Ayumi Shinohara. Compact bit encoding schemes for simply-typed lambda-terms. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, pages 146–157. Association for Computing Machinery, September 2016.

[100] John Tromp. Binary lambda calculus and combinatory logic. In *Kolmogorov Complexity and Applications, 29.01. – 03.02.2006*, volume 06051 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, January 2006.

[101] Raymond Wan. *Browsing and searching compressed documents*. PhD thesis, The University of Melbourne, December 2003.

[102] Raymond Wan. Re-Pair and Des-Pair. `https://github.com/rwanwork/Re-Pair`, Accessed: October 26, 2018.

[103] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes (2nd Ed.): Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers Inc., 2nd edition, December 1999.

[104] Kazuya Yaguchi, Naoki Kobayashi, and Ayumi Shinohara. Efficient algorithm and coding for higher-order compression. In *2014 Data Compression Conference, DCC 2014, Snowbird, UT, USA, 26–28 March, 2014* [5], page 434.

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

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