UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

A STUDY ON REPETITIVENESS MEASURES FOR STRINGS

TESIS PARA OPTAR AL GRADO DE
DOCTOR EN COMPUTACIÓN

CRISTIAN URBINA

PROFESOR GUÍA:
GONZALO NAVARRO

MIEMBROS DE LA COMISIÓN:
HIDEO BANNAI
CRISTIÁN RIVEROS
MATÍAS TORO

Este trabajo ha sido parcialmente financiado por ANID-Subdirección de Capital Humano/Doctorado Nacional/2021-21210580, Fondos Basales FB0001 y AFB240001, Proyecto Fondecyt 1-230755, y Beca NIC Chile.

SANTIAGO DE CHILE
2025

Resumen

En áreas como la Bioinformática, muchas veces es necesario manejar colecciones de datos altamente repetitivos. Por ejemplo, dos genomas humanos comparten cerca del 99.9% de su contenido. Existe la necesidad, tanto para la ciencia como para la industria, de mantener estas enormes colecciones de texto en forma comprimida. Compresores tradicionales basados en la *entropía de Shannon* no son adecuados cuando los datos son repetitivos, pues únicamente explotan las frecuencias de los símbolos. Por esta razón, encontrar medidas de repetitividad y también compresores que la exploten se ha vuelto un problema de investigación relevante.

En esta tesis estudiamos distintas medidas de repetitividad desde un punto de vista matemático y algorítmico: sus propiedades combinatoriales, sus limitaciones, eficiencia de compresores relacionados, y también sus relaciones con otras medidas de repetitividad.

Comenzamos estudiando propiedades del número de corridas de un mismo símbolo en la *transformada de Burrows-Wheeler* de un texto. Analizamos cuánto puede cambiar esta medida al insertar, eliminar, o sustituir un solo símbolo. También mostramos resultados similares para la operación más compleja de aplicar un morfismo sobre un texto. Estos resultados son relevantes, pues en ciertos contextos los datos pueden cambiar en el tiempo.

También exploramos otras formas de explotar la repetitividad en textos. Primero, presentamos una manera de utilizar los llamados *sistemas de Lindenmayer* (*L-systems*), los cuales se basan en morfismos, como representaciones comprimidas, y estudiamos propiedades de su medida asociada. Más aún, combinamos los L-systems con técnicas mejor establecidas de *copy-paste*, dando como resultado los *NU-systems*, los cuales mostramos que son más poderosos que otros compresores que explotan solo mecanismos de copy-paste.

Aunque el tamaño de los L-systems y NU-systems puede ser mucho menor que δ — una cota inferior estable para la repetitividad— en escenarios relevantes, la tarea de proveer acceso directo para ellos en tiempo *polilogarítmico* ha sido elusiva. Por esto, introducimos una extensión de las gramáticas libres de contexto (GLC), llamada *iterated straight-line programs (ISLPs)*, las cuales pueden ser más pequeñas que δ y a la vez proveer acceso directo eficiente. También presentamos un método de balanceo para generalizaciones de GLCs, para simplificar la tarea de brindar acceso directo a posibles extensiones futuras de GLCs.

Nuestra última contribución consiste en extender las medidas de repetitividad para textos regulares a textos *bidimensionales*. Existen muchos tipos de datos en dos dimensiones, como matrices, grafos y coordenadas, entre otros. Estos tipos de datos son abundantes en áreas de la ciencia como la Astronomía, y también pueden ser de naturaleza altamente repetitiva.

Abstract

In areas like Bioinformatics, it is often necessary to handle big collections of highly repetitive data. For example, two human genomes share 99.9% of their content. There is a need in science and industry to maintain those huge string collections in compressed form. Traditional compressors based exclusively on *Shannon's entropy* are not suitable for repetitive data, as they only exploit bias in symbol frequencies. Finding good measures of repetitiveness and also compressors exploiting this repetitiveness has then become a relevant research problem.

In this thesis, we study repetitiveness measures from a mathematical and algorithmic point of view: their combinatorial properties, their limits, the efficiency of the related compressors, and their relationships with other repetitiveness measures.

We start by studying combinatorial properties of the number of *equal-letter runs* of the *Burrows-Wheeler transform* of the text. We show how much this measure can change after performing an insertion, deletion, or substitution of a single symbol. We exhibit similar results for the more complex operation of string morphism application. These results are relevant because in several contexts data can change over time.

We also explore alternative ways to exploit repetitiveness on strings. We find a way to use so-called *Lindenmayer systems (L-systems)*, which build on string morphisms, as compressed representations of strings, and study the properties of its associated repetitiveness measure. Further, we combine L-systems with more established *copy-paste* techniques yielding the *NU-systems*, which we show to be considerably more powerful in terms of space than all the other state of the art compressors exploiting only copy-paste mechanisms.

While the size of L-systems and NU-systems can be much smaller than the measure δ —which is considered to be stable lower-bound for repetitiveness— in some relevant scenarios, the task of providing direct access on them in polylogarithmic time has been elusive. We introduce then an extension of context-free grammars (CFGs), called *iterated straight-line programs (ISLPs)*, which can also be smaller than δ and support efficient direct access. We also provide a method for balancing some general extensions of CFGs, with the aim of simplifying the task of providing direct access on them, if another relevant extension were devised in the future.

Our final contribution is extending repetitiveness measures from regular strings to *two-dimensional strings*. There exist many types of data that are two-dimensional, like matrices, graphs, coordinates, and so on. These types of data are abundant in fields like Astronomy, and can also be very repetitive in nature.

Thank you for reading.

Acknowledgments

My Ph. D. journey is finally coming to its end. Overall, I have enjoyed the trip, and want to thank all the people that has made this experience an enjoyable one.

I start by thanking the people that motivated me to pursue a Ph. D. in the first place. I thank my friends Mara Malewski and Francisco Olivares for their valuable, long lasting and symbiotic friendship. I thank professors Pablo Pérez-Lantero, Rodrigo Abarzúa, Rosa Barrera, Alexis Rojas, and Mónica Soto, for encouraging me to pursue a postgraduate degree.

I thank my supervisor Gonzalo Navarro for his unconditional support during all these years. He gave me a lot of freedom to work, and I always felt trusted, which I really appreciate. He was always up for discussion; gave me a lot of feedback on my work; and helped me with whatever I needed. I hope we might keep working together in the future.

I am also grateful to the people from the Department of Computer Science (DCC) of the University of Chile: Sandra Gáez, Angélica Aguirre, Paz Zañartu, and many others. They helped me a lot when I needed it, and answered all my questions about the Ph.D. program.

I thank professors Marinella Sciortino and Gabriele Fici, for hosting me at the University of Palermo, not once, but two times. They gave me the opportunity to work in some really interesting topics. I also thank my friend Giuseppe Romana, who I met there at University of Palermo. All of them were very kind and supportive. I think we work very well together.

I also thank the nice people and friends I met by attending conferences: Luca Parmigiani, Zsuzsanna Lipták, Cecilia Hernández, Fernanda Sanchirico, Ariel Cáceres, Albani Olivieri, Alejandro Pacheco, Diego Ortego, Josefa Robert, Lorenzo Carfagna, and many others.

I must give a special mention to my family: my parents Veronica and Cristian, my siblings Fabian and Noelia; my dogs Ada, Fox, and Choquita; and my mother's cats Horacio, Julieta, Horacia and Fea. There are also my friends from my hometown: Gonzalo, Javier, Victor, Lucas, Bryan, María José, Pamela, and others. They bring some fun to my life.

During my second stay at Palermo, I met a lot of people that changed my perspective on many topics. I thank Richard, Martín, Chloé, Andy, Yulyana, Ali, Jasmine, Axel, Dhruv, Shahed, Benny, and many others for their valuable friendship and insights on life. I owe a special thank to my friends Diletta and Jayne for always being there when I needed them.

Finally, I thank Éric Tanter, Matías Toro, Cristián Riveros and Hideo Bannai, for agreeing to participate in my Ph.D. defense, and giving me valuable feedback.

Contents

List of Figures

Chapter 1

Introduction

In this chapter we define the motivation, scope, and overall structure of the thesis. We also highlight the main contributions of this work.

The outline of this chapter is as follows.

- In Section 1.1, we present a brief motivation for studying repetitiveness measures.
- In Section 1.2, we formally state the thesis goals.
- In Section 1.3, we present our main contributions, mentioning the articles where these results appeared, and how the author of this thesis was involved in them.
- In Section 1.4, we give a general outline of the remaining chapters of the manuscript.
- Finally, in Section 1.5, we explain and clarify the notations and conventions used along the manuscript.

1.1 Motivation

In areas like Bioinformatics, it is often necessary to handle big collections of highly repetitive data. For example, two human genomes share 99.9% of their content [115]. In another scenario, for sequencing a genome, one extracts so-called *reads* (short substrings) from it, with a “coverage” of up to 100X, which means that each position appears on average in 100 reads.[1] There is a need in science and industry to maintain those huge string collections in compressed form. Traditional compressors based exclusively on *Shannon's entropy* are not suitable for repetitive data, as they only exploit bias in symbol frequencies for compressing. Finding good measures of repetitiveness and also compressors exploiting this repetitiveness has then become a relevant research problem.

[1] https://www.illumina.com/science/technology/next-generation-sequencing/plan-experiments/coverage.html

There is an increasing interest in (1) defining measures of compressibility (i.e., functions from strings to $\mathbb{R}$ aiming to quantify the compressibility of strings) that are useful for highly repetitive texts, (2) developing compressed text representations whose size can be bounded in terms of those measures, and (3) providing efficient (i.e., polylogarithmic time) access methods to those compressed texts, so that algorithms can be run on them without ever decompressing the texts [99, 100]. We call *lower-bounding measures* those satisfying (1), *reachable measures* those (asymptotically) reached by the size of a compressed representation (2), and *accessible measures* those reached by the size of representations satisfying (3).

For example, the size γ of the smallest "string attractor" of a text T —γ and the other measures in this paragraph will be properly defined in upcoming chapters— is a lower bounding measure, unknown to be reachable [71], and smaller than the size reached by all known compressors exploiting repetitiveness. The size b of the smallest "bidirectional macro scheme" of T [127], and the size z of the Lempel-Ziv parse of T [85], are the reachable measures with the best theoretical and observed performance in terms of space on highly repetitive texts, with z being computable in linear time and b being NP-hard to compute. The size $g_{\mathtt{rl}}$ of the smallest run-length context-free grammar generating (only) T is the smallest accessible measure to date, though NP-hard to compute. It holds $\gamma \leq b \leq z \leq g_{\mathtt{rl}}$ for every text.

One of the most attractive lower bounding measures devised so far is δ [79]. This measure has several nice properties: it can be computed in linear time and lower bounds all previous measures of compressibility, including γ, for every text. While δ is known to be unreachable, the measure $\delta^* = \delta \log \frac{n \log \sigma}{\delta \log n}$, where σ is the size of the alphabet, retains many of its good properties, on top of being reachable: $\Omega(\delta^*)$ is the space needed to represent some text family for each n, σ, and δ; within $\mathcal{O}(\delta^*)$ space it is possible to represent every text T and access any length-λ substring of T in time $\mathcal{O}(\lambda + \log n)$ [79], together with more powerful operations [79, 78, 69]. On the other hand, δ^* is not a lower bound for most repetitiveness measures, as δ is. In particular, it holds that $g_{\mathtt{rl}} = \mathcal{O}(\delta^*)$.

There is plenty of room for studying compressibility measures for highly repetitive texts, as it is still a relatively new line of research. For improving the current state of the art on repetitiveness measures —which basically means finding smaller reachable measures, or finding accessible measures whose size is smaller than $g_{\mathtt{rl}}$— it is necessary to study them from a mathematical and algorithmic point of view: their combinatorial properties, their limitations, the efficiency of their related compressors, and their relationships with other repetitiveness measures.

Exploring alternative ways to exploit repetitiveness on strings is also fundamental, as most of the state of the art compressors and measures of repetitiveness are based exclusively on *copy-paste* techniques (i.e., techniques considering almost exclusively explicit copies of substrings as source of repetitiveness), which could be a limiting factor. There could be other types of regularities that have not been explored yet.

Another open flank regarding measuring repetitiveness is how to do it when the data is presented in other formats. For instance, there exist many types of data that are *two-dimensional*, like matrices, graphs, coordinates, and so on. These types of data are abundant in fields like Astronomy, and can also be very repetitive in nature. And yet, not many advances have been made in this matter in the latest years.

1.2 Thesis Statement

We focus on building a solid foundation and intuition on how to handle repetitiveness. More specifically, we

1. study state of the art repetitiveness measures from a combinatorial point of view;
2. design novel repetitiveness measures for strings that can help us to understand the limitations of state of the art repetitiveness measures;
3. generalize repetitiveness measures for strings to data structured in other ways, like matrices.

We firmly believe that to find better ways of handling the huge volumes of data and query requirements of today's world, it is fundamental to build a strong foundation on repetitiveness measures and their limitations.

1.3 Contribution

The content of this manuscript has been retrieved mostly from articles on which the author of this thesis has worked on. We present these articles grouped by topics and then sorted by date of publication.

The first group of articles contains the following.

[60] S. Giuliani, S. Inenaga, Z. Lipták, G. Romana, M. Sciortino, and C. Urbina. Bit catastrophes for the Burrows-Wheeler transform. In *Proc. 27th International Conference on Developments in Language Theory (DLT 2023)*, volume 13911 of *Lecture Notes in Computer Science*, pages 86–99. Springer, 2023.

[42] G. Fici, G. Romana, M. Sciortino, and C. Urbina. On the impact of morphisms on BWT-runs. In *Proc. 34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023)*, volume 259 of *Leibniz International Proceedings in Informatics*, pages 10:1–10:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[61] S. Giuliani, S. Inenaga, Z. Lipták, G. Romana, M. Sciortino, and C. Urbina. Bit catastrophes for the burrows-wheeler transform. *Theory of Computing Systems*, 69(2):19, 2025.

In these articles we present new combinatorial properties of the famous *Burrows-Wheeler transform*, with respect to edit operations and morphism application. In concrete, the author of this thesis mostly worked on lower bounding the additive sensitivity of the number of *BWT-runs* to edit operations, introducing the notions of *abelian order-preserving morphism* and *abelian order-reversing morphism*, and characterizing the BWT of binary words after the application of *Thue-Morse-like morphisms* and the *period-doubling morphism*.

The second group of articles contains the following.

[105] G. Navarro and C. Urbina. On stricter reachable repetitiveness measures. In *Proc. 28th International Symposium on String Processing and Information Retrieval (SPIRE 2021)*, volume 12944 of *Lecture Notes in Computer Science*, pages 193–206. Springer, 2021.

[106] G. Navarro and C. Urbina. L-systems for measuring repetitiveness. In *Proc. 34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023)*, volume 259 of *Leibniz International Proceedings in Informatics*, pages 25:1–25:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[108] G. Navarro and C. Urbina. Repetitiveness measures based on string morphisms. *Theoretical Computer Science*, 1043:115259, 2025.

In these articles we approach the task of measuring repetitiveness in strings using string morphisms. We show several properties of our new defined measures, and establish asymptotic relations with other state of the art measures.

The third group of articles contains the following.

[103] G. Navarro, F. Olivares, and C. Urbina. Balancing run-length straight-line programs. In *Proc. 29th International Symposium on String Processing and Information Retrieval (SPIRE 2022)*, volume 13617 of *Lecture Notes in Computer Science*, pages 117–131. Springer, 2022.

[107] G. Navarro and C. Urbina. Iterated straight-line programs. In *Proc. 16th Latin American Theoretical Informatics Symposium (LATIN 2024)*, volume 14578 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2024.

[104] G. Navarro, F. Olivares, and C. Urbina. Generalized straight-line programs. *Acta Informatica*, 62(1):14, 2025.

In these articles we extend grammar compression techniques and known balancing procedures to a more general framework. In concrete, the author of this thesis mostly contributed on extending context-free grammars to *generalized straight-line programs* (GSLPs), adapting a well-known balancing procedure for SLPs [52] to work on GSLPs, specifying GSLPs to *iterated straight-line programs* (ISLPs), and providing direct access to arbitrary positions of the text using ISLPs.

Finally, we have the following article.

[24] L. Carfagna, G. Manzini, G. Romana, M. Sciortino, and C. Urbina. Generalization of repetitiveness measures for two-dimensional strings. In *Proc. 31th International Symposium on String Processing and Information Retrieval (SPIRE 2024)*, volume 14899 of *Lecture Notes in Computer Science*, pages 57–72. Springer, 2024

In this article we work on the problem of extending repetitiveness measures to higher dimensional spaces. The author of this thesis contributed on defining bidimensional versions of state of the art repetitiveness measures, establishing separations between them, as well as showing that linearizations techniques might be suboptimal as a compression booster for 2D strings.

1.4 Structure of the Thesis

The structure of this thesis is as follows.

- In Chapter 2, we present some basic concepts from the field of *Combinatorics on words* used throughout this thesis.
- In Chapter 3, we presents some basic concepts of the field of *Text compression and indexing*.
- In Chapter 4, we focus on the notion of *repetitiveness*. In particular, we focus on how we can exploit this repetitiveness to achieve high compression rates. We present state of the art repetitiveness measures, and asymptotic relations among them.
- In Chapter 5 we study one of the most famous and widely used tools in Text Compression: the Burrows-Wheeler transform (BWT). The BWT is a permutation of the symbols in a text, and it is often used to boost compressibility. When a text is highly repetitive, the BWT tends to have long equal-letter runs, making it suitable for run-length compression. The size of the run-length encoding of the BWT is a widely used measure of repetitiveness. In concrete, we study how much the number of BWT equal-letter runs can vary after the application of an *edit operation* like insertion, deletion and substitution; or the more complex operation of *morphism application*.
- In Chapter 6, we explore alternative ways to exploit repetitiveness on strings. We find a way to use so-called *Lindenmayer systems* (L-systems), which build on string morphisms, as compressed representations of strings, and study the properties of its associated repetitiveness measure ℓ, which is always smaller than the size g of the smallest context-free grammar generating (only) the text.

  We combine L-systems with more established *copy-paste* techniques and define *NU-systems* (and its associated measure ν), which we show to be considerably more powerful in terms of space than all the other state of the art compressors exploiting only copy-paste mechanisms.

  We show that ℓ and ν, for some string families, can break the better established measure δ based on substring complexity, a lower bound for most measures and compressors exploiting repetitiveness.

  Further, we show that by preprocessing some variants of the L-systems, we can provide direct access to arbitrary positions of the text in logarithmic time. No strong upper bounds were found for the size of these variants, though.

- In Chapter 7, we generalize a recent result that states that *Straight-Line Programs (SLPs)* can be balanced, to a general class of grammars we call *Generalized SLPs (GSLPs).* We then specialize GSLPs to so-called *Iterated SLPs (ISLPs).*

  We prove that ISLPs break, for some text families, the measure δ. Further, ISLPs can support efficient extraction of substrings. This was the first compressed representation for repetitive texts of size upper bounded by the size $g_{\mathtt{rl}}$ of the smallest *Run-Length Straight-Line Program (RLSLP)*, breaking δ in some string families, and at the same time, supporting direct access to arbitrary text symbols in polylogarithmic time.

- In Chapter 8, we extend some repetitiveness measures for strings to *two-dimensional strings*, i.e., matrices with their entries over an alphabet Σ. This is important because 2D strings are a fairly common form of data, an they can be highly repetitive.

  We start by proving some results relating these new measures. We also show that there are important differences in these relations with respect to the 1D setting.

  Finally, we show that a systematic study specific to higher dimensional strings is necessary, as applying one-dimensional compressors on 2D strings after the use of some standard linearization technique can lead in some cases to poor compressibility.

- In Chapter 9, we summarize our contributions, and present some open questions, and further lines of research.

1.5 Notation and Conventions

We clarify some notations and conventions utilized in the thesis manuscript.

Strings/words/texts: We utilize variables a, b, c, d and others to refer to arbitrary symbols or letters. We utilize variables w, x, y, z, u, v and others to refer to arbitrary strings. We use **boldface** to emphasize that a string is infinite by the right, e.g. $\mathbf{w} = \mathtt{aaa}\cdots$. For concrete letters, we use `typewritter font`, e.g. `abracadabra` is a word on the alphabet $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{r}\}$. When considering topics within the fields of Text Compression and Text Indexing, we usually refer to strings as *texts*, and use T, S and other capital letters to denote them.

Numbers: We focus mainly on discrete problems, so $p, q, n, m, i, j, k, t, r$ are usually natural numbers or integers depending on the context. If the base of a logarithm is not specified, $\log n$ stands for the logarithm in base 2 of n.

Computational model: To measure the space needed to store a value (e.g. a string or an integer), and also to measure the time utilized by algorithms, we assume the *transdichotomous RAM model of computation* [46]. In this model, the space utilized is measured in words of $\Theta(\log n)$ bits, where n is the size of the input (typically, of the uncompressed string or text). Access to a specific word, arithmetical and logical operations are assumed to take constant time.

Chapter 2

Combinatorics on Words

In this chapter, we introduce some terminology[1], concepts and elementary results in *Combinatorics on Words*, that are needed to understand the rest of thesis.

The outline of the chapter is as follows.

- In Section 2.1, we present some basic definitions on strings, and provide some examples to illustrate them.
- In Section 2.2, we explore one of the most elementary ideas studied in Combinatorics on Words, the notion of *primitiveness*. We also present a fundamental theorem characterizing primitive words, by Lyndon and Schützenberger.
- In Section 2.3, we explain what a *string morphism* is, and also exhibit some relevant highly repetitive infinite words that can be defined by using them, usually called *morphic words*.
- Finally, in Section 2.4, we introduce the family of *Sturmian words*, i.e., aperiodic infinite binary words of minimal complexity, which have been widely studied in the Combinatorics on words field, and also have been used to prove separations between repetitiveness measures.

2.1 Basic Concepts

We explain some basic concepts on strings.

Alphabets and finite strings. An (ordered) *alphabet* is a finite set of *symbols* $\Sigma = \{a_1, \dots, a_\sigma\}$ extended with a total order $a_1 < \cdots < a_\sigma$. A (finite) *string* w is a finite

[1] There are some differences in how the areas of Text Compression and Combinatorics on Words refer to the same objects. For instance, in Text Compression it is usual to refer to a sequence of symbols as a *string*, or a *text*. In Combinatorics on Words, strings are simply called *words*. Our choice is to use the most appropriate term and notation depending on the context.

sequence $w[1]w[2]\cdots w[n]$ of symbols where $w[i] \in \Sigma$ for $i \in [1\,..\,n]$, and its *length* is denoted by $|w| = n$. The unique *empty string*, whose length is 0, is denoted by ε. The set of all finite strings over Σ is denoted by Σ^*. The set of non-empty finite strings is denoted by Σ^+. We let $\mathsf{alph}(w) \subseteq \Sigma$ be the set of letters actually appearing in w.

Substrings, prefixes and suffixes. Let $x = x[1]\cdots x[n]$ and $y = y[1]\cdots y[m]$ be strings; the concatenation operation $x \cdot y$ (or just xy) yields the string $x[1]\cdots x[n]\,y[1]\cdots y[m]$. We denote by w^k the concatenation of w with itself, k times. Let $w = xyz$. Then y (resp., x, z) is a *substring* (resp., *prefix*, *suffix*) of w. It is *proper* if it is not equal to w, and *non-trivial* if it is distinct from ε and w. The notation $w[i\,..\,j]$ stands for the substring $w[i]w[i+1]\cdots w[j]$ if $i \leq j$, and ε otherwise. We also use the conventions $w[i\,..\,j] = w[..\,j] = w[1\,..\,j]$ if $i < 1$, and $w[i\,..\,j] = w[i\,..] = w[i\,..\,n]$ if $j > n$. Substrings are sometimes called *factors*.

Run-length encoding. The *run-length encoding* of a string $w[1\,..\,n]$ is a sequence of pairs $\mathtt{rle}(w) = (a_1, p_1), (a_2, p_2), \ldots, (a_k, p_k)$ where each a_i is a single symbol, each p_i is a positive integer, and such that $w = a_1^{p_1} a_2^{p_2} \cdots a_k^{p_k}$ and $a_i \neq a_{i+1}$ for $i \in [1\,..\,(k-1)]$. E.g. $\mathtt{rle}(\mathtt{aaaabaaabb}) = (\mathtt{a}, 4), (\mathtt{b}, 1), (\mathtt{a}, 3), (\mathtt{b}, 2)$.

Reverse of a string and palindromes. The *reverse* of a string $w[1\,..\,n]$ is defined as the string $w^R = w[n] \cdot w[n-1] \cdots w[1]$. A string w such that $w = w^R$ is said to be a *palindrome*. E.g. the word `racecar` is a palindrome.

Rotations of a string The *i-th rotation* (or *conjugate*) of w is the string $\mathtt{rot}_w(i) = w[i\,..\,n] \cdot w[1\,..\,(i-1)]$ for each $i \in [1\,..\,n]$. The *multiset of rotations* $\mathcal{R}(w)$ contains all the (possibly repeated) rotations of w. $\mathcal{R}_x(w)$ contains all the possibly repeated rotations of w starting with the prefix x. E.g. $\mathcal{R}(\mathtt{abaa}) = \{\mathtt{aaab}, \mathtt{aaba}, \mathtt{abaa}, \mathtt{baaa}\}$ and $\mathcal{R}_{\mathtt{aa}}(\mathtt{abaa}) = \{\mathtt{aaab}, \mathtt{aaba}\}$.

Infinite strings. A (right) *infinite string* $\mathbf{w}$ (we use boldface to emphasize them) over an alphabet Σ is a mapping from $\mathbb{Z}^+$ to Σ. The length of an infinite string is denoted ω, which is greater than n for any $n \in \mathbb{Z}^+$. The concatenation $x \cdot \mathbf{y}$ is well-defined when x is finite and $\mathbf{y}$ infinite, as $x \cdot \mathbf{y} = x[1]\cdots x[|x|]\mathbf{y}[1]\mathbf{y}[2]\cdots$. The definitions of substring, prefix, and suffix carry over to infinite strings. Note that proper prefixes of infinite strings are always finite strings, and suffixes are always infinite strings. The notations $\mathbf{w}[i]$, $\mathbf{w}[i\,..\,j]$ and $\mathbf{w}[i\,..] = \mathbf{w}[i]\mathbf{w}[i+1]\cdots$ also carry over to infinite strings.

Lexicographic order Let $<$ be the total order relation among the symbols in Σ. Let x and y be (possibly infinite) strings. The *lexicographic order* among strings is defined inductively as follows: i) No string is lexicographically smaller than itself. ii) The empty string ε is assumed to be lexicographically smaller than any other string. iii) For non-empty strings the lexicographic order is defined by the relation $x < y$ if and only if $x = ax'$ and $y = by'$ with $a < b$. or $a = b$ and $x' < y'$.

Factor complexity of strings. Let $F(w)$ be the set of distinct substrings of w, where w may be infinite or not. The *factor complexity function* of w counts the number of different substring in w for each possible length k. Formally, $P_w(k) = |\{x \in F(w) \mid |x| = k\}|$.

Edit operations, edit distance, and Hamming distance Let $w = xy$ be a finite string with x, y strings, and a and b symbols. Then $w' = xay$ can be obtained from w via an *insertion*. Similarly, if $w = xay$, then $w' = xy$ and $w' = xby$ can be obtained via a *deletion* and a *substitution*, respectively. These three operations are called *edit operations*. As a shortcut, if $w[1 \mathinner{.\,.} n]$, then $\widehat{w} = w[1 \mathinner{.\,.} n-1]$.

The *edit distance* $d_{\texttt{edit}}(x, y)$ between two finite strings x and y counts the minimum number of edit operations needed to transform x into y, or vice versa (both numbers coincide). If x and y have the same length, their *Hamming distance*, which counts the number of *mismatches* between them, is $d_H(x, y) = |\{i \in [1 \mathinner{.\,.} |x|] \mid x[i] \neq y[i]\}|$.

Symbol frequencies, Parikh vectors and balancedness. We denote by $|w|_a$ the number of as appearing in w. Moreover, if x is a string, we let $|w|_x$ be the number of possibly overlapping occurrences of x within w. The *Parikh vector* of a word w is defined as $\texttt{parikh}(w) = (|w|_{a_1}, |w|_{a_2}, \ldots, |w|_{a_\sigma})$, that is, the Parikh vector encompasses the frequencies of each symbol in w.

A (possibly infinite) word w is *balanced* if for every two substring x and y of the same length and any symbol $a \in \Sigma$, it holds $||x|_a - |y|_a| \leq 1$, that is, for each symbol, their frequency in x and y almost coincides. A finite word w is *circularly balanced* if all its rotations are balanced. E.g. the word `abaababa` is circularly balanced.

2.2 Primitive Words, Powers, and Lyndon Words

We introduce the notion of *primitiveness*.

Definition 2.2.1 A string $w = a_1 \cdots a_n$ is *periodic* with *period* p if $a_i = a_{i+p}$ for $i \in [1 \mathinner{.\,.} n-p]$. If n/p is an integer, then w is a *power* of $a_1 \ldots a_p$, that is, $w = (a_1 \ldots a_p)^{n/p}$. A word w is *primitive* if $w = u^k$ implies that $k = 1$.

Simply put, primitive words are those that cannot be written as a power of another word.

Example 2.2.2 The word `abaab` is primitive. In fact, any word of length p with p a prime number is either primitive, or it is equal to a^p, for some symbol a. On the other hand, the word `abaaba` is not primitive, because it can be written as $(\texttt{aba})^2$.

Similar notions are defined for infinite words.

Definition 2.2.3 An infinite word $\mathbf{w}$ is *periodic* with *period* p if $a_i = a_{i+p}$ for all $i \in \mathbb{Z}^+$. It is *ultimately periodic* if it can be written as $\mathbf{w} = x \cdot \mathbf{y}$ with $\mathbf{y}$ periodic. If an infinite word is not ultimately periodic, then it is *aperiodic*.

Example 2.2.4 The word $\mathbf{w} = \mathtt{aba}^\omega$ is ultimately periodic, but not periodic. The sequence of digits of any irrational number is an aperiodic infinite word.

One key result concerning primitiveness and powers is the following lemma by Lyndon and Schützenberger [91].

Lemma 2.2.5 (Lyndon and Schützenberger) Two words $x, y \in \Sigma^+$ commute, i.e., $xy = yx$, if and only if they are powers of the same word z, i.e., $x = z^p$ and $y = z^q$ for some $p, q > 0$.

From this lemma, some equivalent characterizations of primitive words can be derived.

Lemma 2.2.6 ([114]) Let $w = a_1 \ldots a_n$. The following conditions are equivalent:

1. w is primitive.
2. $w \neq xy$ for any two non-empty commuting words x and y.
3. w has n distinct rotations.

Some basic results on primitiveness are the following.

Lemma 2.2.7 Any word w can be written uniquely as $w = x^p$ for some primitive word x and some positive integer p.

Lemma 2.2.8 A word $w = x^p$ with x primitive and p a positive integer has $|x|$ distinct rotations.

Among primitive words, there is a relevant subclass that has been widely studied.

Definition 2.2.9 A *Lyndon word* is a word w that is primitive and lexicographically strictly smaller than any of its rotations.

Example 2.2.10 The word `aaaaab` is Lyndon, whereas its rotation `aaaaba` is not. Also, the word `ababab` is not Lyndon because it is not primitive.

2.3 String Morphisms

The set Σ^* together with the (associative) concatenation operator and the (identity) string ε form a *monoid* structure $(\Sigma^*, \cdot, \varepsilon)$. A *morphism* on strings is a function $\varphi : \Sigma_1^* \to \Sigma_2^*$ satisfying $\varphi(x \cdot y) = \varphi(x) \cdot \varphi(y)$ for all x and y (i.e., a function preserving the monoid structure), where Σ_1 and Σ_2 are arbitrary alphabets. To define a morphism on strings, it is sufficient to define how it acts over the symbols in its domain. The pairs $(a, \varphi(a))$ for $a \in \Sigma_1$, usually denoted $a \to \varphi(a)$, are called the *rules* of the morphism, and there are $|\Sigma_1|$

of them. If $\Sigma_1 = \Sigma_2$, then the morphism is called an *endomorphism*. We sometimes use the notation $\varphi \equiv (\varphi(a_1), \ldots, \varphi(a_\sigma))$ to describe the action of a morphism on the letters of Σ. The notation $\varphi^i(x)$, as it is usual with functions, denotes the iteration of φ starting on x, i times. The notation $\varphi^\omega(x)$ stands for $\lim_{i\to\infty} \varphi^i(x)$ if the limit exists.

Let $\varphi : \Sigma_1^* \to \Sigma_2^*$ be a morphism on strings. Some useful definitions are $\texttt{width}(\varphi) = \max_{a\in\Sigma_1} |\varphi(a)|$ and $\texttt{size}(\varphi) = \sum_{a\in\Sigma_1} |\varphi(a)|$. A morphism is *non-erasing* if $\forall a \in \Sigma_1, |\varphi(a)| > 0$, *expanding* if $\forall a \in \Sigma_1, |\varphi(a)| > 1$, *k-uniform* if $\forall a \in \Sigma_1, |\varphi(a)| = k > 1$, and it is a *coding* if $\forall a \in \Sigma_1, |\varphi(a)| = 1$ (i.e., it is 1-uniform).

Morphisms can be used to define infinite sequences with low factor complexity (precisely, with $p_{\mathbf{w}}(n) = \mathcal{O}(n^2)$), among some other regularity properties. We are interested in studying these kind of sequences, as low factor complexity generally implies a high degree of repetitiveness.

Definition 2.3.1 Let $\varphi : \Sigma^* \to \Sigma^*$ be a string morphism. Then, φ is *prolongable* on a symbol a if $\varphi(a) = ax$ for some string $x \neq \varepsilon$.

If a morphism φ is prolongable on a symbol a, then for each i, j with $0 \leq i \leq j$, it holds that $\varphi^i(a)$ is a prefix of $\varphi^j(a)$. Moreover, the infinite word

$$\mathbf{w} = \varphi^\omega(a) = ax\varphi(x)\varphi^2(x)\cdots$$

is well defined, and it is the unique infinite fixed-point of φ starting with the symbol a [90].

By considering the fixed-points of prolongable morphisms, some relevant classes of infinite words arise.

Definition 2.3.2 Let $\varphi : \Sigma^* \to \Sigma^*$ be a string morphism prolongable on a symbol $a \in \Sigma$, and let $\tau : \Sigma \to \Gamma$ be a coding. Then,

1. the fixed-point $\varphi^\omega(a)$ is called a *purely morphic word*;
2. the image $\tau(\varphi^\omega(a))$ is called a *morphic word*;
3. if φ is k-uniform for some $k > 1$, the image $\tau(\varphi^\omega(a))$ is called an *automatic word*.

A particularly relevant family of words is defined by the morphism $\Phi \equiv (\texttt{ab}, \texttt{a})$ known as the *Fibonacci morphism*. The finite words $\Phi^i(\texttt{a})$ are called *finite Fibonacci words*, and their limit $\Phi^\omega(\texttt{a})$ is a purely morphic word called the *Fibonacci word*.

The class of morphic words properly contains both the class of purely morphic words and the class of automatic words. On the other hand, neither the class of purely morphic words nor the class of automatic words is a subset one of the other. A recent survey shows detailed examples on this matter [2].

As we already foreshadowed, the classes of morphic words, purely morphic words, and automatic words are limited in the number of distinct factors they can contain. It has been proven that the number of different factors of length n appearing in these kinds of words is $\mathcal{O}(n^2)$, $\mathcal{O}(n \log n)$, and $\mathcal{O}(n)$, respectively [39].

2.4 Sturmian Words

One of the most studied string families in Combinatorics on Words are the so-called *Sturmian words* [90]. They will prove useful for us because of their high degree of repetitiveness, and many properties. They are defined as follows.

Definition 2.4.1 An infinite word $\mathbf{w}$ is said to be *Sturmian* if $P_{\mathbf{w}}(n) = n + 1$ for all $n \geq 0$.

Note that, because $P_{\mathbf{w}}(1) = 2$, Sturmian words are infinite binary words. Hence, in the following we assume $\Sigma = \{\mathtt{a}, \mathtt{b}\}$.

It is known that if there exists some n_0 such that $P_{\mathbf{w}}(n_0) \leq n_0$ then $\mathbf{w}$ is periodic [90]. In this sense, Sturmian words are aperiodic infinite binary words with minimal factor complexity. There are many other equivalent definitions of Sturmian words [90]. Among them, the following one will be particularly useful for us in Chapter 5.

Theorem 2.4.2 Let $\mathbf{w}$ be an infinite binary word. Then, $\mathbf{w}$ is Sturmian if and only if it is aperiodic and balanced.

Among Sturmian words, a special class can be easily constructed algorithmically [90].

Definition 2.4.3 Given an infinite sequence of integers $d_0, d_1, d_2, \ldots$, with $d_0 \geq 0, d_i > 0$ for all $i > 0$, called *directive sequence*, we define their associated *standard Sturmian words* as $s_0 = \mathtt{b}$, $s_1 = \mathtt{a}$, and $s_{i+1} = s_i^{d_{i-1}} s_{i-1}$, for $i \geq 1$. An infinite word $\mathbf{w}$ is said to be a *characteristic Sturmian word* if it is the limit of an infinite sequence of standard Sturmian words, i.e., $\mathbf{w} = \lim_{i \to \infty} s_i$.

We illustrate the above definition and construction in Example 2.4.4.

Example 2.4.4 Consider the directive sequence $1, 1, 1, \ldots$. The words $s_0 = \mathtt{b}$, $s_1 = \mathtt{a}$, $s_2 = \mathtt{ab}$, $s_3 = \mathtt{aba}$, $s_4 = \mathtt{abaab}$, and so on, are exactly (with the exception of s_0) the finite Fibonacci words. The limit of this sequence of words is the infinite Fibonacci word $\mathtt{abaababaabaab} \cdots$. Thus, the Fibonacci word is a characteristic Sturmian word.

Chapter 3

Text Compression and Indexing

In this chapter we present the basic concepts within the field of *Text Compression and Indexing* needed to understand the following chapters.

The chapter is structured as follows:

- In Section 3.1, we introduce some basic definitions regarding compressibility measures.
- In Section 3.2, we present a widely used compressibility measure known as *empirical entropy*, and show why it does not work so well on repetitive datasets.
- In Section 3.3, we present the basic queries to be answered on text collections.
- In Section 3.4, we introduce the concepts of additive and multiplicative sensitivity of compressibility measures to string transformations.

3.1 Compressibility Measures

We start by introducing a formalization of the notions of *compressor* and *compressibility measure*.

Definition 3.1.1 A *lossless text compressor*[1] (or just *compressor*) is an algorithm $C : \Sigma^* \rightarrow \Gamma^*$ such that there exists a *decompression algorithm* D for which $D(C(w)) = w$, for any $w \in \Sigma^*$. The string $C(w)$ is called a *compressed representation* of w.

Definition 3.1.2 A *compressibility measure* μ is a function $\mu : \Sigma^* \rightarrow \mathbb{N}$.

For a compressibility measure to be meaningful or useful, it should capture the degree of compressibility of strings in some way. That is, the more compressible is a string w, the

[1] In this work we focus only on *lossless* text compression, that is, the compressed representation together with the decompression algorithm must always allow us to recover the original text. This is different from *lossy compression*, which is used for instance, for compressing images. In this type of compression, some information might be permanently lost after the compression process.

smaller the value $\mu(w)$ should be. Naturally, the strings for which a compressibility measure takes a low value are highly dependent on the particular features of the texts to be exploited.

Definition 3.1.3 We say that a measure μ is *reachable* if we can represent every string $w[1\,..\,n]$ within $\mathcal{O}(\mu(w))$ space (where the asymptotics refer to n).

Example 3.1.4 Let C be a compressor, and $\mu_C(w) = |C(w)|$. The measure μ_C is reachable by definition, as we can represent any string in that space by using the compressor C.

We measure space in $\Theta(\log n)$-bit words following the conventions of the transdichotomous RAM model of computation. Hence, $\mathcal{O}(\mu(w))$ space means $\mathcal{O}(\mu(w)\log n)$ bits. We can represent any symbol in the alphabet of $w[1\,..\,n]$ using a constant number of words as long as $|\Sigma| = \mathcal{O}(n^d)$ for some constant $d \geq 0$.

We introduce some terminology that will come handy when comparing compressibility measures in terms of their asymptotic behavior.

Definition 3.1.5 A compressibility measure μ_1 *is smaller* or *lower bounds* another compressibility measure μ_2 if $\mu_1(w) = \mathcal{O}(\mu_2(w))$ for every $w[1\,..\,n] \in \Sigma^*$. If, in addition, there is an infinite string family $\mathcal{F} \subseteq \Sigma^*$ where $\mu_1(w) = o(\mu_2(w))$ for every $w[1\,..\,n] \in \mathcal{F}$, we say that μ_1 *is strictly smaller* or *strictly lower bounds* μ_2. Two compressibility measures μ_1 and μ_2 are *equivalent* if each one lower bounds the other, and *uncomparable* if $\mu_1 = o(\mu_2)$ on a string family $\mathcal{F}_1$ and $\mu_2 = o(\mu_1)$ on another string family $\mathcal{F}_2$.

Compressibility measures remove the implementation noise from compressors, allowing a simpler mathematical analysis in terms of their space usage. When measures are based on a compressor, they allow us to understand how well these compressors can perform on certain domains, or how sensitive these compressors are to string transformations. They also allow us to compare the performance of different compressors in a simplified way. Some measures which are not based on actual compressors can be used to upper-bound or lower-bound other measures, which is useful to provide theoretical guarantees on the performance of theoretical and practical compressors.

3.2 Empirical Entropy

A fundamental notion in text compression is *Kolmogorov's complexity.* Essentially, this measure quantifies the minimum space needed to represent any given string in a computer, with the representation being a computer program that outputs the string. We define it in terms of Turing machines as follows.

Definition 3.2.1 The Kolmogorov's complexity $\mathcal{K}(w)$ of a string w, is the number of bits needed to describe the Turing machine with the shortest description, that outputs w when run on an empty tape.

While Kolmogorov's complexity is the ideal measure of compressibility in terms of space, it is uncomputable, that is, we cannot get it using a computer program that always halts.

Theorem 3.2.2 ([30]) It is undecidable for a string w and an integer k if $\mathcal{K}(w) \leq k$.

The problem of Kolmogorov's complexity being unpractical has motivated the design of a variety of compressors trying to approximate this value. They do so by exploiting the regularities that are most likely to be present on the texts.

One of the most practical computable measures of compressibility is the *empirical entropy* of the text. It is defined as follows.

Definition 3.2.3 The *zero-order empirical entropy* of a string $w[1\mathinner{.\,.}n]$ is

$$\mathcal{H}_0(w) = \sum_{a\in\Sigma} \frac{|w|_a}{n} \log \frac{n}{|w|_a}.$$

When the only sources of compressibility to be exploited are the relative frequencies of symbols, the zero-order empirical entropy is a tight lower bound to the output size of any compressor.

If instead, contexts of length k for some fixed k are believed to be a better predictor for the following symbol, the *k-th order empirical entropy* of the text might be a better choice.

Definition 3.2.4 The *k-th order empirical entropy* of the a string $w[1\mathinner{.\,.}n]$ is

$$\mathcal{H}_k(w) = \sum_{x\in\Sigma^k} \frac{|w|_x}{n} \mathcal{H}_0(w_x),$$

where w_x is the string formed by the characters following x in w. Empirical entropy has been and still is the base of many widely used text compressors like *gzip*[2] and *bzip2*[3]. These are usually called *statistical compressors* [10].

Empirical entropy and statistical compressors do not perform well when considering highly repetitive strings. Consider any string $w[1\mathinner{.\,.}n]$. One can notice that for any $k > 0$, it holds that $\mathcal{H}_0(w) = \mathcal{H}_0(w^k)$, as the relative frequencies for each symbol in $\mathtt{alph}(w)$ are the same in both w and w^k. In other words, for both strings we need the same amount of bits to represent their symbols. This means that any compressor reaching zero-order empirical entropy needs $\Theta(n\mathcal{H}_0(w))$ bits to represent w and $\Theta(kn\mathcal{H}_0(w))$ bits to represent w^k. A similar analysis holds for H_k [81].

A better alternative to represent w^k would be to encode it as a pair (w, k). To do so, we might encode w in $\Theta(n\mathcal{H}_0(w))$ bits using some statistical compressor, and then encode k using $\mathcal{O}(\log k)$ bits with any basic integer representation. This sums to a total of $\mathcal{O}(\log(k) + n\mathcal{H}_0(w))$ bits, which can be orders of magnitude smaller than what a naive statistical compressor may achieve when applied on w^k.

[2] `https://www.gnu.org/software/gzip/manual/gzip.html`

[3] `https://sourceware.org/bzip2/`

In Chapter 4, we present compressibility measures and compressors that are able to detect this kind of redundancies, thereby performing better than $\mathcal{H}_0$ and $\mathcal{H}_k$ on text where the degree of repetitiveness is high.

3.3 Text Indexing

There exists many questions one may want to ask about a text. Let $T[1\mathinner{.\,.}n]$ and $P[1\mathinner{.\,.}m]$ be two strings with $m \leq n$. The most relevant queries under the scope of this thesis are the following.

- $\texttt{access}(T, i)$: Outputs the symbol $T[i]$, with $i \in [1\mathinner{.\,.}n]$.
- $\texttt{extract}(T, i, j)$: Outputs the string $T[i\mathinner{.\,.}j]$, with $i, j \in [1\mathinner{.\,.}n]$.
- $\texttt{locate}(T, P)$: Outputs a list containing the starting position of each occurrence of the pattern $P[1\mathinner{.\,.}m]$ within $T[1\mathinner{.\,.}n]$.
- $\texttt{count}(T, P)$: Outputs the number of distinct occurrences of the pattern $P[1\mathinner{.\,.}m]$ within $T[1\mathinner{.\,.}n]$.

Example 3.3.1 Let a text $T = \texttt{a}\underline{\texttt{bra}}\texttt{cada}\underline{\texttt{bra}}$ and a pattern $P = \texttt{bra}$. Then, $\texttt{access}(T, 5) = \texttt{c}$, $\texttt{extract}(T, 4, 8) = \texttt{acada}$, $\texttt{locate}(T, P) = \{2, 9\}$ and $\texttt{count}(T, P) = 2$.

When both T and P are stored in uncompressed form, the *optimal time* to answer direct access queries is $\mathcal{O}(1)$. Using $\mathcal{O}(|T|)$ extra space to store some data structures on top of T, one can answer locate and count in $\mathcal{O}(m + |T|_P)$ and $\mathcal{O}(m)$ time, respectively, which is *optimal* as it is always necessary to read the pattern, and in the case of $\texttt{locate}(T, p)$, we also need to report the occurrences of P.

On the other hand, when either the text T, the pattern P, or both are presented in compressed form, the times given above to answer these queries are not necessarily achievable. For instance, Verbin and Yu [128] showed that when using some compression techniques based on context-free grammars, access always takes $\Omega(\log n / \log\log n)$ time.

Therefore, when dealing with compressed representations reaching $\mathcal{O}(\mu)$ space for some compressibility measure μ, is it usual to focus first in reaching *efficient time* for these queries, as optimal time might be difficult to achieve. We understand by efficient time any time that is at most $\mathcal{O}(\operatorname{polylog} n)$ times worse than the optimal time.

Definition 3.3.2 The measure μ is *accessible* if we can answer $\texttt{access}(T, i)$ in $\mathcal{O}(\operatorname{polylog}(n))$ time using $\mathcal{O}(\mu)$ space.

3.4 Sensitivity to String Transformations

When dealing with datasets that might change over time, common sense is that it would be practical to have compressed representations that may be easily be updated after an edit operation on the original dataset. That is, starting from a compressed representation $C(w)$ of a string w, it should be fast to obtain a compressed representation $C(w')$ of w' where $d_{\texttt{edit}}(w, w') = 1$. Naturally, this can only be satisfied if $|C(w')|$ is close enough to $|C(w)|$, otherwise just outputting the new representation would take considerable time.

In this thesis, we consider a simpler related problem. We are interested in measuring how much a compressibility measure can vary after applying an edit operation, or more generally, a string transformation, to the original text. Naturally, reachable measures that do not change much after an edit operation are good candidates to perform well in an online setting, though finding a way to perform the update is another problem.

The change on a measure after applying a string transformation on a text can be quantified either additively or multiplicatively by taking the difference or the ratio, respectively. Given a measure and a string transformation, we define their *additive sensitivity function* and *multiplicative sensitivity function* as follows.

Definition 3.4.1 Let $\mu : \Sigma^* \to \mathbb{R}^+$ be a compressibility measure, and $\texttt{op} : \Sigma^* \to \Sigma^*$ be any string transformation. Then,

$$\begin{aligned} AS_{\mu,\texttt{op}}(n) &= \max_{w \in \Sigma^n} \{|\mu(\texttt{op}(w)) - \mu(w)|\} \text{ and} \\ MS_{\mu,\texttt{op}}(n) &= \max_{w \in \Sigma^n} \{\mu(\texttt{op}(w))/\mu(w)\}. \end{aligned}$$

These definitions have already been introduced for edit operations [1], though for these it is customary to consider the set of all possible insertions/substitutions/deletions instead of just a fixed one. Sensitivity has also be defined for the reverse operation [59].

In general, we are interested in upper bounding and lower bounding these two functions. For instance, if $AS_{\mu,\texttt{op}}(n) = \mathcal{O}(1)$, this means that μ is highly resistant to the transformation $\texttt{op}$, as μ can increase only by a constant.

On the other hand, the multiplicative sensitivity is more useful to show that a measure is highly sensitive to some string transformation, by finding a growing lower bound to this function.

Another useful concept when analyzing measures is the notion of *monotonicity*.

Definition 3.4.2 A measure μ is *monotone* upon a string operation $\texttt{op}$ if for any string w it holds $\mu(\texttt{op}(w)) \geq \mu(w)$.

Chapter 4

Measuring Repetitiveness

As we showed in Chapter 3, statistical compressors whose space is lower bounded by the empirical entropy of the text, do not scale well when considering powers of the same text. That is, statistical compressors cannot detect and exploit the simplest form of repetitiveness one can think of. This observation makes clear the need of designing text compressors and compressibility measures that are suitable for scenarios where the data is highly repetitive. The prime examples of this kind of data are the enormous genome collections used in Bioinformatics.

We will say that a compressibility measure μ is a *repetitiveness measure*, if it (arguably) captures the degree of *repetitiveness* of strings. The more repetitive is a string w, the smaller the value $\mu(w)$ should be. In general, a repetitive string is understood as one containing many copies of the same substrings, but there is no single agreed-upon measure of repetitiveness.

In this chapter, we present and explain the most relevant repetitiveness measures to be considered in this thesis. We illustrate those we work with at a deeper level. The chapter is structured as follows.

- In Section 4.1, we present repetitiveness measures based on *parsings* like the size of the famous *Lempel-Ziv parse* of the text and its variants; and others like *bidirectional macro schemes.*
- In Section 4.2, we present repetitiveness measures based on *context-free grammars.*
- In Section 4.3, we introduce the *Burrows-Wheeler transform*, which yields a permutation of the symbols of the text, and two closely related repetitiveness measures based on the number of equal-letter runs of this transform.
- In Section 4.4, we explain the concept of a *string attractor*, and introduce the measure γ based on it, which is a lower bound to all reachable measures presented before.
- In Section 4.5, we explain how the substring complexity function is used to define a repetitiveness measure called δ, which lower bounds γ and is considered a lower bound for repetitiveness.

- In Section 4.6, we end with a brief summary of the asymptotic relations between repetitiveness measures.

All of the measures, except δ and (possibly) γ, are reachable because they are defined as the size of some compression method; in all those cases the represented string $w[1\,..\,n]$ can be decompressed in optimal time, $\mathcal{O}(n)$. More in-depth surveys on repetitiveness measures exist [99, 100].

4.1 Parsing-based Measures

A *parsing* of size k is a factorization of a string w into non-empty *phrases*, $w = w_1 \cdot w_2 \cdots w_k$ where $w_i \in \Sigma^+$ for $1 \leq i \leq k$. Several compressors work by parsing w in a way that just some summary information about the phrases enables recovering w.

The *Lempel-Ziv (LZ) parsing* [85] processes a string greedily from left to right, always forming the longest phrase that has a copy (called a *source*) starting inside some previous phrase, or else forming an *explicit* phrase of length 1. Lempel-Ziv compression encodes non-explicit phrases as pairs (p, l), where p indicates where the source starts in w and l is the phrase length. In LZ, the source can overlap the new phrase. The *LZ-no parsing*, instead, does not allow the source overlap the new phrase. The *LZ-end parsing* [80] requires, in addition, that the source ends at a previous phrase boundary. All of these parsings can be constructed in linear time [85, 70], and their number of phrases are denoted by z, $z_{\texttt{no}}$, and $z_{\texttt{e}}$, respectively. While z and $z_{\texttt{no}}$ are optimal among the parsings satisfying their respective conditions, this is not always the case for $z_{\texttt{e}}$. The size of the *optimal LZ-end parsing*, i.e., a factorization where each phrase w_{i+1} appears as a suffix of $w_1 \ldots w_j$ for some $j \leq i$, is denoted by $z_{\texttt{end}}$, and it is NP-hard to compute [7]. Because of the optimality of z, $z_{\texttt{no}}$, and $z_{\texttt{end}}$, it holds that $z \leq z_{\texttt{no}} \leq z_{\texttt{end}} \leq z_{\texttt{e}}$ for every string. We illustrate the differences between the LZ variants in Example 4.1.1.

Example 4.1.1 Consider the string $w = \texttt{abracadabracadabracadabra}$. Then, we have

$$\begin{aligned} LZ(w) &= \texttt{a} \cdot \texttt{b} \cdot \texttt{r} \cdot \texttt{a} \cdot \texttt{c} \cdot \texttt{a} \cdot \texttt{d} \cdot \texttt{abracadabracadabra} \\ LZ_{\texttt{no}}(w) &= \texttt{a} \cdot \texttt{b} \cdot \texttt{r} \cdot \texttt{a} \cdot \texttt{c} \cdot \texttt{a} \cdot \texttt{d} \cdot \texttt{abracad} \cdot \texttt{abracadabra} \\ LZ_{\texttt{e}}(w) &= \texttt{a} \cdot \texttt{b} \cdot \texttt{r} \cdot \texttt{a} \cdot \texttt{c} \cdot \texttt{a} \cdot \texttt{d} \cdot \texttt{abracad} \cdot \texttt{abracad} \cdot \texttt{abra} \end{aligned}$$

where the source of every non-explicit phrase is the first position. Hence, $z(w) = 8$, $z_{\texttt{no}}(w) = 9$, and $z_{\texttt{e}}(w) = 10$.

A *bidirectional macro scheme* (BMS) [127] is any parsing where each phrase of length greater than 1 has a copy starting at a different position (to its left or to its right) in such a way that the original string can be recovered by following these pointers (assuming that the phrases of length 1 store their symbol explicitly). The measure $b(w)$ is defined as the size of the smallest BMS for w. It strictly lower bounds all the other reachable repetitiveness measures [102], except for the ones we define in this thesis. Computing $b(w)$ is NP-hard [51].

Example 4.1.2 Let $w = \texttt{abaababaabaababaababaabaababaabaab}$. A BMS for w is

$$\texttt{abaababaabaababaaba} \cdot \texttt{b} \cdot \texttt{a} \cdot \texttt{abaababaabaab}.$$

The source of the first phrase begins at position 14 and ends at position 32, the second and third phrase are explicit symbols, and the source of the fourth phrase begins at position 14 and ends at position 26. Note how the source of the first phrase is to its right, and the source of the fourth phrase is to its left. Hence, $b(w) \leq 4$.

Another interesting parsing-based measure is the size of the greedy *lexicographic parsing* of w, denoted as $v(w)$ [102]. This parsing processes w from left to right, taking as the next phrase the longest common prefix between the unprocessed part of the string and its lexicographically smaller suffix (a unique symbol $, smaller than the others, is assumed to exist at the end of w). It forms an explicit phrase of length one if the longest common prefix is empty or no predecessor exists. It has been proven that b strictly lower bounds v [102].

4.2 Grammar-based Measures

A *context-free grammar (CFG)* is a 4-tuple $G = (V, \Sigma, R, S)$, where V is a set of symbols called the *variables*, Σ is an alphabet of *terminals* such that $V \cap \Sigma = \emptyset$, $R \subseteq V \times (V \cup \Sigma)^*$ is called the set of *rules*, and $S \in V$ is the *initial variable*. For the sake of readability, we write the rules (A, x) as $A \rightarrow x$. Let $u, v \in (V \cup \Sigma)^*$. If $A \rightarrow x$ is a rule, then uAv *yields* uxv, denoted as $uAv \Rightarrow uxv$. We say that u *derives* v if $u \Rightarrow^* v$, where $\Rightarrow^*$ is the reflexive-transitive closure of the relation $\Rightarrow$. The *language* generated by the grammar G is $L(G) = \{w \in \Sigma^* \,|\, S \Rightarrow^* w\}$, that is, the strings of terminals that can be derived from the initial variable.

A *straight-line program (SLP)* is a CFG G such that for any variable there is exactly one rule, which can be either a *terminal rule* $A \rightarrow a$ with $a \in \Sigma$, or a *binary rule* $A \rightarrow BC$ where $B, C \in V$, and satisfying that for each $A \in V$ and $u \in (V \cup \Sigma)^*$, if $A \Rightarrow^* u$, then A does not occur in u, that is, there are no cycles in the derivation of the grammar. These conditions ensure that the language of the SLP G is a singleton $L(G) = \{w\}$.

Since there are no cycles, the variables of an SLP can always be given a total order, so that if $A \rightarrow BC$, then $B, C < A$. We only consider CFGs with one rule per variable, that admit such a total order, though the right-hand sides of rules may have zero or more terminals and variables. That is, if $A \rightarrow \alpha B \beta$ is a rule, then $B < A$. Such CFGs are guaranteed to generate a unique string, which is denoted $\texttt{exp}(G) = w$. We extend this notation to the unique strings generated by the variables of the grammar.

The *size* of a CFG $G = (V, \Sigma, R, S)$ is $\texttt{size}(G) = \sum\{|x| \mid A \rightarrow x \in R\}$, the sum of the lengths of the right-hand sides of its rules. The repetitiveness measure $g(w)$ is defined as the least size of a CFG G generating w. Computing $g(w)$ is NP-hard [121, 31], though there exist log-approximations [63, 121].

Another measure related to CFGs that strictly lower bounds $g(w)$ is $g_{\texttt{rl}}(w)$, the least size of a *run-length CFG (RLCFG)* generating w [110]. RLCFGs extend CFGs by allowing

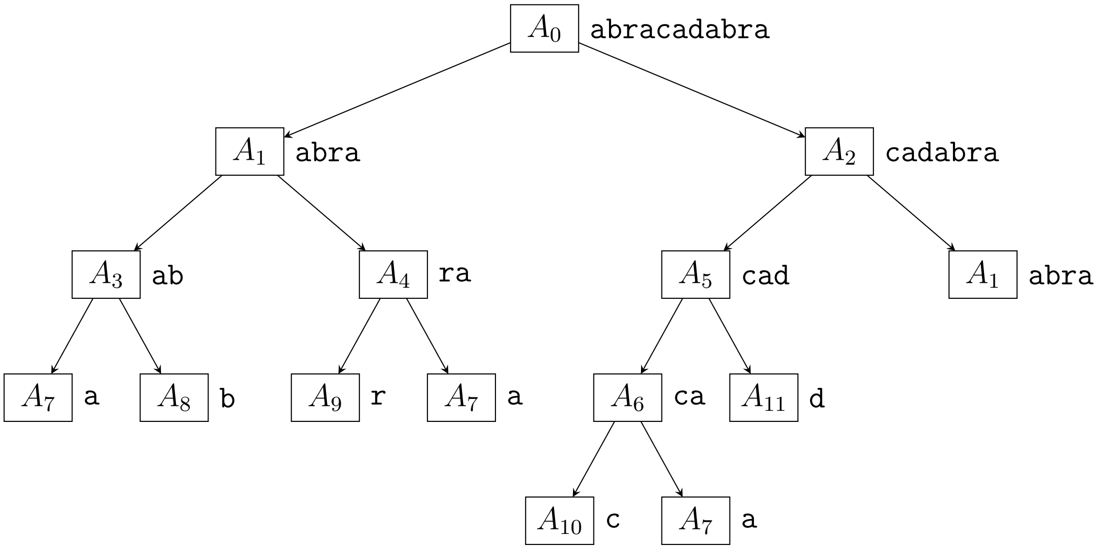


Figure 4.1: Grammar tree of an SLP G generating the word `abracadabra`. The grammar is defined by the binary rules $A_0 \to A_1A_2$, $A_1 \to A_3A_4$, $A_2 \to A_5A_1$, $A_3 \to A_7A_8$, $A_4 \to A_9A_7$, $A_5 \to A_6A_{11}$, $A_6 \to A_{10}A_7$, and the terminal rules $A_7 \to \mathtt{a}$, $A_8 \to \mathtt{b}$, $A_9 \to \mathtt{r}$, $A_{10} \to \mathtt{c}$, $A_{11} \to \mathtt{d}$. At the right of each node there is the expansion of each variable. The height of the SLP is 4, and its size is 19

constant-size rules of the form $A \to B^k$ for $k > 1$ and $B \in V$, and we again consider only RLCFGs that follow a total order in their variables. RLCFGs can be a log-factor smaller than CFGs in some string families like $\{\mathtt{a}^n \mid n \geq 0\}$, where $g = \Theta(\log n)$ and $g_{\mathtt{rl}} = \mathcal{O}(1)$.

Some useful notions related to SLPs and RLSLPs are the following. The *derivation or parse tree* of an SLP is an ordinal tree where the nodes are the variables, the root is the initial variable, and the leaves are the terminal variables. The children of a node are the variables appearing in the right-hand side of its rule (in left-to-right order). The *height* of an SLP or RLSLP is the length of the longest path from the root to a leaf node in its derivation tree. The derivation tree of RLSLPs is analogous to that of SLPs; the nodes labeled A, for the rules $A \to B^t$, have t children labeled B. The *grammar tree* is obtained by pruning the parse tree so that only the leftmost occurrence of a nonterminal is retained as an internal node and all the others become leaves. Rules $A \to B^t$ are represented as the node A having a left child B (which can be internal or a leaf) and a special right child denoting B^{t-1} (which is a leaf). The size of the grammar tree is proportional to the size of the grammar. We show an example in Figure 4.1.

Composition-systems [53] extend CFGs with constant-size *extraction* rules of the form $A \to B[i : j]$ for some $i, j \in [1 \mathinner{.\,.} |\mathtt{exp}(B)|]$, which mean that $\mathtt{exp}(A) = \mathtt{exp}(B)[i \mathinner{.\,.} j]$. Still, the symbols must be ordered and it must hold $B < A$ for such a rule to be valid. *Collage-systems* [73] extend CFGs with run-length rules and extractions, thereby combining composition-systems and RLCFGs. The size $c(w)$ of the smallest collage-system deriving w lower bounds z [102], while z strictly lower bounds $g_{\mathtt{rl}}$ [16].

	i	m	i	s	s	i	s	s	i	p	p
	i	p	p	i	m	i	s	s	i	s	s
	i	s	s	i	p	p	i	m	i	s	s
	i	s	s	i	s	s	i	p	p	i	m
$I = 5$	m	i	s	s	i	s	s	i	p	p	i
	p	i	m	i	s	s	i	s	s	i	p
	p	p	i	m	i	s	s	i	s	s	i
	s	i	p	p	i	m	i	s	s	i	s
	s	i	s	s	i	p	p	i	m	i	s
	s	s	i	p	p	i	m	i	s	s	i
	s	s	i	s	s	i	p	p	i	m	i

Figure 4.2: BWT matrix of the string $w = \texttt{mississippi}$. The row where w appears is colored in gray. Its run-length encoding is $(\texttt{p}, 1)(\texttt{s}, 2)(\texttt{m}, 1)(\texttt{i}, 1)(\texttt{p}, 1)(\texttt{i}, 1)(\texttt{s}, 2)(\texttt{i}, 2)$, hence $r(w) = 8$.

4.3 Burrows-Wheeler Transform

The *Burrows-Wheeler transform* (BWT) is a reversible transformation of a word, which usually makes it more compressible [21]. The combinatorial formulation of the BWT consists in sorting all the rotations (repeated rotations can appear multiple times) of the word lexicographically, and then concatenating the last symbol of each rotation, in order.

Definition 4.3.1 Let w be a word of length n, and $w_1, w_2, \ldots, w_n$ be the sequence of rotations of w in lexicographic order. Then, $\texttt{bwt}(w) = w_1[n]w_2[n] \cdots w_n[n]$.

A useful concept when analyzing the BWT of a text is its *BWT matrix*.

Definition 4.3.2 Let w be a word of length n. The *BWT matrix* of w is a square matrix M of dimension n, whose cells are defined as $m_{i,j} = w_i[j]$, where w_i is the i-th rotation of w in lexicographic order. The index of the occurrence of w in the sequence of sorted conjugates is denoted by I.

In Figure 4.2 it can be seen the BWT-matrix of the word mississippi.

Though the length of $\texttt{bwt}(w)$ is the same as the length of w, the run-length encoding of $\texttt{bwt}(w)$ tends to be orders of magnitude smaller when the string is highly repetitive. Intuitively, this happens because if a substring ax of w is repeated many times, the rotations starting with x will contain many as as their last symbols. The proximity and abundance of these as make it more likely to form long clusters of the same symbol in the BWT.

In Text Indexing, it is customary to consider the run-length encoding of the $-terminated version of the BWT, because of its relation with the suffix tree data structure. Nevertheless, the size of the run-length encoding of the BWT can vary wildly depending on whether this $ is appended or not at the end of the string [60]. Hence, we consider them two different measures. We introduce their definitions.

Definition 4.3.3 Let $w \in \Sigma$ and assume $\$ < a$ for all $a \in \Sigma$. Then, $r(w) = \texttt{rle}(\texttt{bwt}(w))$ and $r_{\$}(w) = \texttt{rle}(\texttt{bwt}(w\$))$.

On of the reasons of why $r_{\$}$ and r are widely studied is that there exist indexes that can answer `locate` and `count` in $\mathcal{O}(r_{\$})$ space [50].

Regarding the relation of r and $r_{\$}$ with other measures, it has been proved that $b \leq 2r_{\$} = \mathcal{O}(r_{\$})$ [102, Theorem 9]. A similar result can be easily derived for r, from [102, Theorem 9] and the following lemma.

Lemma 4.3.4 ([119, Lemma 16]) Let $w \in \Sigma^+$ be a Lyndon word. Then, for all $u, v \in \Sigma^*$ such that $w = uv$, it holds $r(vu) + 1 \leq r_{\$}(w) \leq r(vu) + 2$.

Proposition 4.3.5 It always holds that $b = \mathcal{O}(r)$.

Proof. Let $w' = vu \in \Sigma^+$ and $w = uv$ be its Lyndon rotation (i.e., the smallest rotation of w' in lexicographic order). By Lemma 4.3.4, it holds $r(w') = \Theta(r_{\$}(w))$. Therefore, as $b(w) \leq 2r_{\$}(w)$ [102], there exists a BMS for w of size $\mathcal{O}(r(w'))$. This macro scheme can be transformed into a macro scheme for w' of at most twice the size, by reorganizing the phrases, and then splitting as needed those phrases whose source previously started in the prefix u and ended in the suffix v of w. □

BWT based measures are uncomparable to reachable measures other than b; and v in the case of $r_{\$}$, for which it holds $v = \mathcal{O}(r_{\$})$ [102]. It is known that r and $r_{\$}$ are $\Theta(n)$ in circular de Bruijn strings over the binary alphabet [9], while v, g, $z_{\mathtt{e}}$, and even $nH_k/\log n$ (i.e., the statistical entropy measured in words) are $\mathcal{O}(n/\log n)$. On the other hand, in even Fibonacci words, r is $\mathcal{O}(1)$ [95] while $r_{\$}, v$ and c are $\Omega(\log n)$ [102].

4.4 String Attractors

In an attempt of defining a general framework, and finding a suitable lower bound for grammar-based and parsing-based compressors, Kempa and Prezza introduced the notion of a *string attractor* [71].

Definition 4.4.1 A *string attractor* for a text $w[1\,..\,n]$ is a set of positions $\Gamma \subseteq [1\,..\,n]$ such that for each substring $w[i\,..\,j]$ of w, there exist integers $i', j' \in [1\,..\,n]$ and $k \in \Gamma$, such that $w[i\,..\,j] = w[i'\,..\,j']$ and $i' \leq k \leq j'$.

That is, a string attractor is a set Γ such that every substring of w has a copy covering a position in Γ. By using string attractors, Kempa and Prezza define the following measure.

Definition 4.4.2 We denote by $\gamma(w)$ the size of the smallest string attractor for w.

Example 4.4.3 Consider the string $w = \texttt{abracadabra}$. A string attractor for w is the set $\Gamma = \{1, 2, 3, 5, 7\}$. The set Γ is minimal because by definition its size cannot be smaller than

$|\texttt{alph}(w)|$. Thus, $\gamma(w) = 5$.

In the same work, Kempa and Prezza showed that the problem of finding the smallest k-attractor —i.e., a set of positions such that any substring of length at most k is covered by at least one of these positions— is NP-hard for $k \geq 3$ [71]. Recently, the same result was proved to hold also for $k = 2$ [48].

It is easy to see that γ lower bounds b: we can construct an string attractor of size at most $2b$ from the smallest BMS, by including in the attractor the positions corresponding to an explicit phrase or a phrase border in the BMS. Moreover, Bannai et al. [6] showed that γ is a strict lower bound for b via Thue–Morse words. On the other hand, it is still unknown whether space γ, or even $o(\gamma \log(n/\gamma))$, is reachable.

Similarly to r and $r_{\$}$, the measure γ has attracted much attention in the Combinatorics on Words community because of its simple but elegant mathematical formulation. Mantaci et al. [93] studied many combinatorial properties of string attractors, e.g., they show that γ is not monotonic. They also propose a circular variant of string attractors, which has been recently further studied [118]. Many recent works have focused on characterizing γ for prefixes of relevant infinite words [26, 82, 55, 56, 38].

4.5 The Measure δ

Recently, Kociumaka et al. [116, 79] introduced a repetitiveness measure based on the factor complexity function $P_w(k)$, which counts for each possible k the number of distinct substrings of length k appearing in w. It is defined as follows.

Definition 4.5.1 Let $w[1\,..\,n]$ be a string of length n. The measure δ is

$$\delta(w) = \max_{k \in [1\,..\,n]} P_w(k)/k.$$

In Example 4.5.2 we show a family of strings for which it is easy to compute δ solely from their definition.

Example 4.5.2 Consider the string $w = \texttt{abaababaabaababaababa}$. This string is a finite Fibonacci word, and also a standard Sturmian word, hence it holds $P_w(k) \leq k + 1$ for all $k \in [1\,..\,n]$. Note that the value $P_w(k)/k$ is maximized when $k = 1$. Thus, $\delta(w) = 2$.

The measure δ possesses many desirable properties. First, it has been shown that $\delta = \mathcal{O}(\gamma)$ and there exists string families where $\delta = o(\gamma)$ [79]. This makes δ an strict lower bound for all reachable and unreachable repetitiveness measures to date. Also, it is straightforward that δ is insensitive to reversals, and Akagi et al. [1] proved that δ can only increase by a constant additive value after an edit operation. Moreover, the measure δ can be computed in $\mathcal{O}(n)$ time and space [79]. These properties have made δ the gold standard for measuring repetitiveness.

Nevertheless, the measure δ has some shortcomings. The biggest issue is that $\mathcal{O}(\delta)$ space is unreachable [79]. It has been shown that the related measure $\delta^* = \mathcal{O}(\delta \log \frac{n \log \sigma}{\delta \log n})$ can be reached and shares similar properties [79], although it is not a lower bound for most measures like $z_{\mathtt{e}}$, g, $r_\$$ and r. The measure δ^* is *worst-case optimal*, that is, for every values n, σ and δ there is a string family that needs $\Omega(\delta^*)$ to be represented [79]. It is possible to obtain representations using $\mathcal{O}(\delta^*)$ space in polynomial time, that can answer pattern matching queries in almost-optimal time [79, 78].

The measure δ has lead to some interesting upper bounds on other repetitiveness measures. Some representations of size $\mathcal{O}(\delta^*)$ are run-length grammars [79, 78], hence it follows that $g_{\mathtt{rl}}$ is upper bounded by $\mathcal{O}(\delta \log \frac{n \log \sigma}{\delta \log n}) \subseteq \mathcal{O}(\delta \log \frac{n}{\delta})$. The measure g is upper bounded by $\mathcal{O}(\gamma \log^2 \frac{n}{\gamma}) \subseteq \mathcal{O}(\delta \log^3 \frac{n}{\delta})$ [79, 71]. The measure $r_\$$ is upper bounded by $\mathcal{O}(\delta \log \delta \max(1, \log \frac{n}{\delta \log \delta})) \subseteq \mathcal{O}(\delta \log^2 n)$ [68], and it was recently proved that so is r [119, Corollary 18]. Kempa and Saha [72] showed that $z_{\mathtt{e}} = \mathcal{O}(\delta \log^2 \frac{n}{\delta})$. From these bounds, we conclude that even if the repetitiveness measures presented can vary on some string families, they are still close enough to each other, at most within a polylogarithmic factor (more precisely, an $\mathcal{O}(\log^3 n)$ factor with current knowledge).

4.6 Summary

We summarize the known relations between repetitiveness measures in Figure 4.3.

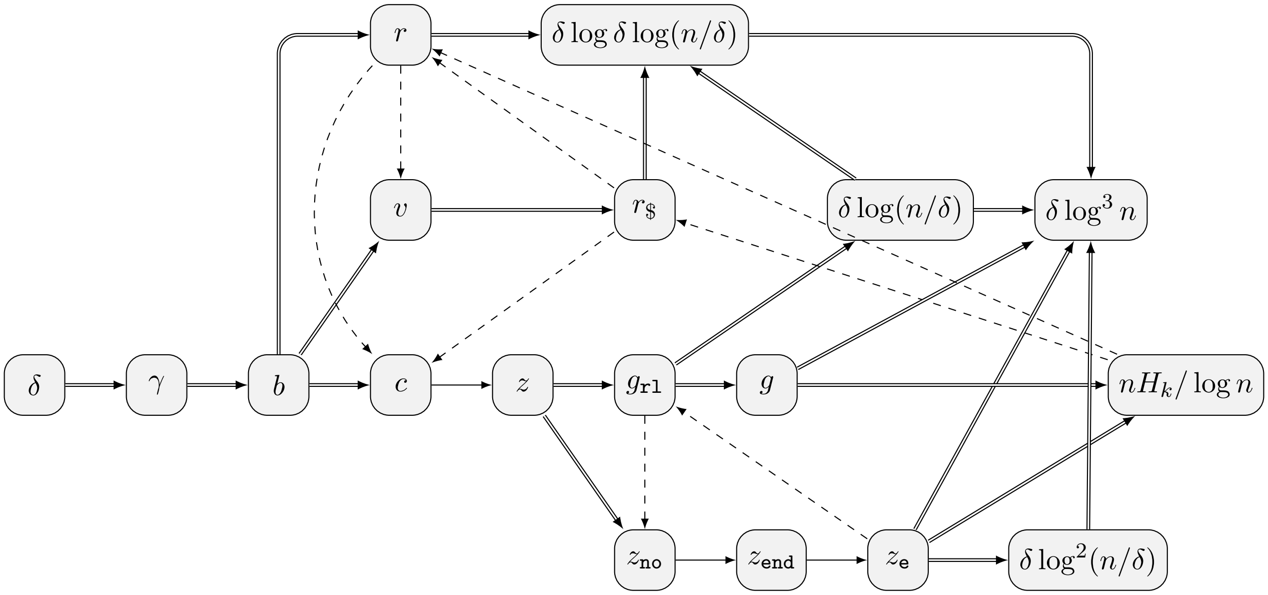


Figure 4.3: Asymptotic relations between state of the art repetitiveness measures. A solid arrow from a measure v_1 to a measure v_2 means that it always holds that $v_1 = \mathcal{O}(v_2)$. A double solid arrow from v_1 to v_2 means that it also exists a string family where $v_1 = o(v_2)$. A dashed arrow from v_1 to v_2 means that there exists a family where $v_1 = o(v_2)$.

In Chapter 6 and Chapter 7 we introduce new repetitiveness measures and relate them to those mentioned in this chapter.

Chapter 5

Sensitivity Properties of the Burrows-Wheeler Transform

In this chapter, we show some results on the sensitivity to string transformations of the measures r and $r_\$$ described in Chapter 4.

Recently, Giuliani et al. [59] showed that the measure r can grow by a $\Theta(\log n)$ multiplicative factor after applying the reverse operation on the input string. In another work, Giuliani et al. [60] showed that the measures r and $r_\$$ can increase by a $\Theta(\log n)$ multiplicative factor after applying any edit operation. Moreover, in the same work the authors show that both measures can increase by a $\Theta(\sqrt{n})$ additive factor after applying any edit operation.

This chapter is structured in two main sections.

- In Section 5.1, we give complete proofs for the results concerning the $\Theta(\sqrt{n})$ additive increase of the measures r and $r_\$$ when applying any edit operation.
- In Section 5.2, we study the impact of morphism applications on the measure r. Though morphism applications are widely used (e.g., they can be used to define many types of codes), they have not been extensively studied regarding their impact on repetitiveness measures. We focus mostly on r, but we also provide results for other repetitiveness measures in order to compare how differently they behave.

5.1 Additive Sensitivity of BWT to Edit Operations

In this section, we exhibit an infinite family of strings on which a single edit operation can cause an additive increment of r by $\Theta(\sqrt{n})$, improving known results on the additive sensitivity of BWT-runs to edit operations. Such a family is defined as follows.

Definition 5.1.1 For any $k > 5$, let $s_i = \mathtt{ab}^i\mathtt{aa}$ and $e_i = \mathtt{ab}^i\mathtt{aba}^{i-2}$ for all $2 \leq i \leq k-1$, and

$q_k = \mathtt{ab}^k\mathtt{a}$. Then,

$$w_k = (\prod_{i=2}^{k-1} s_i e_i) q_k = (\prod_{i=2}^{k-1} \mathtt{ab}^i\mathtt{aaab}^i\mathtt{aba}^{i-2})\mathtt{ab}^k\mathtt{a}.$$

The length of these strings can be easily computed from their definition.

Observation 5.1.2 Let $n = |w_k|$ for some $k > 5$. It holds that $n = \sum_{i=2}^{k-1}(3i+4)+(k+2) = (3k^2+7k-18)/2$. Moreover, it holds that $k = \Theta(\sqrt{n})$.

The following lemma will be used to show how the rotations of w_k can be sorted according to the factorization $s_2 e_2 \cdots s_{k-1} e_{k-1} q_k$.

Lemma 5.1.3 Let $k > 5$ be an integer. Then, $s_2 < e_2 < s_3 < e_3 < \ldots < s_{k-1} < e_{k-1} < q_k$. Moreover the set $\mathcal{U} = \bigcup_{i=2}^{k-1}\{s_i, e_i\} \cup \{q_k\}$ is *prefix-free*, that is, for each two distinct strings in $\mathcal{U}$, none of them is a prefix of the other.

Proof. For the first claim, note from the definition of the strings e_i, s_i and q_k, that for $i \in [2\mathinner{.\,.}k-1]$ it holds $s_i < e_i$, for $i \in [2\mathinner{.\,.}k-2]$ it holds $e_i < s_{i+1}$, and it holds $e_{k-1} < q_k$. For the second claim, observe that for any two distinct strings x and y in the set $\mathcal{U}$ starting with $\mathtt{ab}^j\mathtt{a}$ and $\mathtt{ab}^{j'}\mathtt{a}$ respectively, there are two possible cases. If $j = j'$ then x and y are s_i and e_i respectively, and none of them is a prefix of the other. Otherwise, w.l.o.g. $j < j'$, so $x = \mathtt{ab}^j\mathtt{a}x'$ and $y = \mathtt{ab}^j\mathtt{b}y'$ for some x' and y'. Hence $x[j+2] \neq y[j+2]$ and none of them is a prefix of the other. Thus, the set $\mathcal{U}$ is prefix-free. □

5.1.1 Characterizing the BWT of w_k

In this subsection we characterize the BWT of the word

$$w_k = (\prod_{i=2}^{k-1} s_i e_i) q_k = (\prod_{i=2}^{k-1} \mathtt{ab}^i\mathtt{aa} \cdot \mathtt{ab}^i\mathtt{aba}^{i-2}) \cdot \mathtt{ab}^k\mathtt{a}.$$

To do so, we divide its BWT matrix into disjoint ranges of consecutive rotations sharing the same (specific) prefixes, and characterize the substring of $\mathtt{bwt}(w_k)$ corresponding to each one of these prefixes.

Definition 5.1.4 Given $x, w \in \Sigma^*$, we denote by $\beta(x, w)$ the substring of $\mathtt{bwt}(w)$ corresponding to the range of contiguous rotations prefixed by x. We omit the second parameter of $\beta(x, w)$ when it is clear from the context.

The structure of the whole BWT matrix of w_k is summarized in Figure 5.1. The following series of lemmas characterize the substring of $\mathtt{bwt}(w_k)$ corresponding to each range to be considered.

Lemma 5.1.5 ($\beta(\mathtt{a}^{k-2}\mathtt{b})$) Given the word $w_k = (\prod_{i=2}^{k-1} s_i e_i) q_k$ for some $k > 5$, the first rotation in the BWT matrix is $\mathtt{a}^{k-3} q_k \cdots \mathtt{b}$.

Proof. The first rotation in lexicographic order must start with the longest run of $\mathtt{a}$'s. By definition of w_k, the longest run of $\mathtt{a}$'s has length $k-2$, and it is obtained by concatenating the suffix $\mathtt{a}^{k-3}$ of e_{k-1} with q_k, which is preceded by a $\mathtt{b}$ (otherwise we could extend the run of $\mathtt{a}$'s). □

Lemma 5.1.6 ($\beta(\mathtt{a}^i\mathtt{b})$ for $4 \leq i \leq k-3$) Given the word $w_k = (\prod_{i=2}^{k-1} s_i e_i) q_k$ for some $k > 5$, and an integer $4 \leq i \leq k-3$, the rotations in the BWT matrix starting with $\mathtt{a}^i\mathtt{b}$ are $\mathtt{a}^{i-1}s_{i+2}\cdots\mathtt{b} < \mathtt{a}^{i-1}s_{i+3}\cdots\mathtt{a} < \ldots < \mathtt{a}^{i-1}s_{k-1}\cdots\mathtt{a} < \mathtt{a}^{i-1}q_k\cdots\mathtt{a}$.

Proof. One can notice that, for all $4 \leq i \leq k-3$, the (circular) factor $\mathtt{a}^i\mathtt{b}$ can only be obtained, for all $i+2 \leq j \leq k$, from the concatenation of the suffix $\mathtt{a}^{i-1}$ of e_{j-1}, with either the prefix $\mathtt{ab}$ of s_j, if $i+2 \leq j \leq k-1$, or the prefix $\mathtt{ab}$ of q_k, if $j = k$. By Lemma 5.1.3, we can sort these rotations according to the lexicographic order of $\bigcup_{j=i}^{k-1}\{s_j\} \cup \{q_k\}$. Note that all these rotations end with an $\mathtt{a}$, with the exception of the rotation starting with $\mathtt{a}^{i-1}s_{i+2}$, since it is where the only occurrence of $\mathtt{ba}^i\mathtt{b}$ can be found. □

Lemma 5.1.7 ($\beta(\mathtt{aaab})$) Given the word $w_k = (\prod_{i=2}^{k-1} s_i e_i) q_k$ for some $k > 5$, the first five rotations in the BWT matrix starting with $\mathtt{aaab}$ are $\mathtt{aa}e_2\cdots\mathtt{b} < \mathtt{aa}e_3\cdots\mathtt{b} < \mathtt{aa}e_4\cdots\mathtt{b} < \mathtt{aa}s_5\cdots\mathtt{b} < \mathtt{aa}e_5\cdots\mathtt{b}$, while the remaining are $\mathtt{aa}s_6\cdots\mathtt{a} < \mathtt{aa}e_6\cdots\mathtt{b} < \ldots < \mathtt{aa}s_{k-1}\cdots\mathtt{a} < \mathtt{aa}e_{k-1}\cdots\mathtt{b} < \mathtt{aa}q_k\cdots\mathtt{a}$.

Proof. Analogously to the proof of Lemma 5.1.6, some of the rotations starting with $\mathtt{aaab}$ can be obtained, for all $5 \leq j \leq k$, from the concatenation of the suffix $\mathtt{aa}$ of e_{j-1}, with either the prefix $\mathtt{ab}$ of s_j, if $5 \leq j \leq k-1$, or the prefix $\mathtt{ab}$ of q_k, if $j = k$. However, in this case we have more rotations starting with $\mathtt{aaab}$, that are those rotations starting with the suffix $\mathtt{aa}$ of $s_{j'}$ concatenated with the prefix $\mathtt{ab}$ of $e_{j'}$, for all $2 \leq j' \leq k-1$. Thus, all the rotations starting with $\mathtt{aaab}$ are sorted according to the lexicographic order of the words in $\bigcup_{j=5}^{k-1}\{s_j\} \cup \bigcup_{j'=2}^{k-1}\{e_{j'}\} \cup \{q_k\}$. Note that all the rotations starting either with $\mathtt{aa}s_j$, for all $6 \leq j \leq k-1$, or with $\mathtt{aa}q_k$, end with $\mathtt{a}$. On the other hand, the rotations starting either with $\mathtt{aa}s_5$ or with $\mathtt{aa}e_j$, for all $2 \leq j \leq k-1$, end with a $\mathtt{b}$. □

Lemma 5.1.8 ($\beta(\mathtt{aab})$) Given the word $w_k = (\prod_{i=2}^{k-1} s_i e_i) q_k$ for some $k > 5$, the first five rotations in the BWT matrix starting with $\mathtt{aab}$ are $\mathtt{a}s_2\cdots\mathtt{b} < \mathtt{a}e_2\cdots\mathtt{a} < \mathtt{a}e_3\cdots\mathtt{a} < \mathtt{a}s_4\cdots\mathtt{b} < \mathtt{a}e_4\cdots\mathtt{a}$, while the remaining are $\mathtt{a}s_5\cdots\mathtt{a} < \mathtt{a}e_5\cdots\mathtt{a} < \ldots < \mathtt{a}s_{k-1}\cdots\mathtt{a} < \mathtt{a}e_{k-1}\cdots\mathtt{a} < \mathtt{a}q_k\cdots\mathtt{a}$.

Proof. Each of the rotations starting with $\mathtt{aaab}$ from Lemma 5.1.7 induces a rotation starting with $\mathtt{aab}$, obtained by shifting on the left one character $\mathtt{a}$. It follows that all of these rotations end with an $\mathtt{a}$. The other rotations starting with $\mathtt{aab}$ are the one obtained by concatenating the suffix $\mathtt{a}$ of e_3 and the prefix $\mathtt{ab}$ of s_4, and the one obtained by concatenating the suffix $\mathtt{a}$ of q_k and the prefix $\mathtt{ab}$ of s_2. Moreover, both the rotations end with a $\mathtt{b}$. The thesis follows by sorting the rotations according to the lexicographic order of the words in $\{s_2\} \cup \bigcup_{j=4}^{k-1}\{s_j\} \cup \bigcup_{j'=2}^{k-1}\{e_{j'}\} \cup \{q_k\}$. □

Lemma 5.1.9 ($\beta(\mathtt{ab})$) Given the word $w_k = (\prod_{i=2}^{k-1} s_i e_i) q_k$ for some $k > 5$, the first $k-2$ rotations in the BWT matrix starting with $\mathtt{ab}$ are $\mathtt{aba}^{k-3}q_k\cdots\mathtt{b} < \mathtt{aba}^{k-4}s_{k-1}\cdots\mathtt{b} < \ldots <$

$\mathtt{ab}s_3 \cdots \mathtt{b}$, the following four rotations are $s_2 \cdots \mathtt{a} < e_2 \cdots \mathtt{a} < s_3 \cdots \mathtt{b} < e_3 \cdots \mathtt{a}$, and the remaining are $s_4 \cdots \mathtt{a} < e_4 \cdots \mathtt{a} < \ldots < s_{k-1} \cdots \mathtt{a} < e_{k-1} \cdots \mathtt{a} < q_k \cdots \mathtt{a}$.

Proof. For any two distinct integers $i, i' \geq 0$, we have that $\mathtt{aba}^i\mathtt{b} < \mathtt{aba}^{i'}\mathtt{b}$ if and only if $i > i'$. Thus, the first rotation in lexicographic order starting with $\mathtt{ab}$ is the one which is followed by the longest run of $\mathtt{a}$'s. The smallest of these rotations can be found by concatenating the suffix $\mathtt{aba}^{k-3}$ of e_{k-1} with the prefix $\mathtt{ab}$ of q_k, followed by the suffix $\mathtt{aba}^{i-2}$ of e_{i-1} concatenated with the prefix $\mathtt{ab}$ of s_i, for all $3 \leq i \leq k-1$ taken in decreasing order. By construction of e_i, for all $3 \leq i \leq k-1$, these rotations must end with a $\mathtt{b}$.

The remaining rotations starting with $\mathtt{ab}$ are exactly those rotations having as prefix either s_i or e_i, for all $2 \leq i \leq k-1$, or q_k. Note that all of these rotations are obtained by shifting on the left one character $\mathtt{a}$ from the rotations starting with $\mathtt{aab}$ from Lemma 5.1.8, with the exception of the one starting with s_3. It follows that the latter ends with a $\mathtt{b}$, while all the other rotations with an $\mathtt{a}$. □

Lemma 5.1.10 ($\beta(\mathtt{ba})$) Given the word $w_k = (\prod_{i=2}^{k-1} s_i e_i) q_k$ for some $k > 5$, the first $k-5$ rotations in the BWT matrix starting with $\mathtt{ba}$ are $\mathtt{ba}^{k-3} q_k \cdots \mathtt{a} < \mathtt{ba}^{k-4} s_{k-1} \cdots \mathtt{a} < \ldots < \mathtt{ba}^3 s_6 \cdots \mathtt{a}$, followed by $\mathtt{baa}e_2 \cdots \mathtt{b} < \mathtt{baa}e_3 \cdots \mathtt{b} < \mathtt{baa}e_4 \cdots \mathtt{b} < \mathtt{baa}s_5 \cdots \mathtt{a} < \mathtt{baa}e_5 \cdots \mathtt{b}$, then by $\mathtt{baa}e_6 \cdots \mathtt{b} < \mathtt{baa}e_7 \cdots \mathtt{b} < \ldots < \mathtt{baa}e_{k-1} \cdots \mathtt{b} < \mathtt{ba}s_2 \cdots \mathtt{b} < \mathtt{ba}s_4 \cdots \mathtt{a}$, and finally by $\mathtt{baba}^{k-3} q_k \cdots \mathtt{b} < \mathtt{baba}^{k-4} s_{k-1} \cdots \mathtt{b} < \ldots < \mathtt{bab}s_3 \cdots \mathtt{b} < \mathtt{b}s_3 \cdots \mathtt{a}$.

Proof. One can notice that we have as many circular occurrences of $\mathtt{ba}$ as the number of maximal (circular) runs of $\mathtt{b}$'s in w_k. Then, for all $2 \leq i \leq k-1$, we have (i) one run of $\mathtt{b}$'s in s_i, and (ii) two runs in e_i, and (iii) one run in q_k.

For the case (i), we have one rotation starting with $\mathtt{baa}e_i$, for each $2 \leq i \leq k-1$. Since each run of $\mathtt{b}$'s within each word from $\bigcup_{i=2}^{k-1}\{s_i\}$ is of length at least 2, all rotations in (i) end with a $\mathtt{b}$.

For the case (ii), for all $2 \leq i \leq k-1$, we can distinguish between two sub-cases, based on where $\mathtt{ba}$ starts: if either (ii.a) from the first run of $\mathtt{b}$'s in e_i, or (ii.b) from the second one. For the case (ii.a), we can see that these rotations are of the type $\mathtt{baba}^{i-2} s_{i+1}$, if $2 \leq i < k-2$, and $\mathtt{baba}^{k-3} q_k$. Analogously to the case (i), each rotations for case (ii.a) end with a $\mathtt{b}$. Each rotation in (ii.b) is obtained by shifting two characters on the right each rotation in (ii.a). Therefore, all of these rotations end with an $\mathtt{a}$ and have prefixes of the type $\mathtt{ba}^{i-2} s_{i+1}$, if $2 \leq i < k-2$, or $\mathtt{ba}^{k-3} q_k$.

For the case (iii), the rotation starting with $\mathtt{ba}$ in q_k has $\mathtt{ba}s_2$ as prefix, and it is preceded by a $\mathtt{b}$.

Observe that only for (ii.b) we have rotations starting with $\mathtt{baaaa}$. Hence, the first rotation in lexicographic order is the one starting with $\mathtt{ba}^{k-3} q_k$, followed by those starting with $\mathtt{ba}^{k-4} s_{k-1} < \mathtt{ba}^{k-5} s_{k-2} < \ldots < \mathtt{baaa}s_6$.

Among the remaining rotations, those having prefix $\mathtt{baaa}$ either start with $\mathtt{baa}s_5$ from (ii.b), or $\mathtt{baa}e_i$ from (i), for all $2 \leq i \leq k-1$. Thus, by Lemma 5.1.3, we can sort them according to the order of the words in $\{s_5\} \cup \bigcup_{i=2}^{k-1}\{e_i\}$. Then, the remaining rotations with prefix $\mathtt{baa}$ are those starting with $\mathtt{ba}s_2$ from (iii), and $\mathtt{ba}s_4$ from (ii.b). Finally, let us focus on the rotations from case (ii.a). These rotations are sorted according to the length of the run of $\mathtt{a}$'s following the common prefix $\mathtt{bab}$, similarly to the sorting of the rotations from the case (ii.b). The last rotation left is the one starting with $\mathtt{b}s_3$ from case (ii.b). Since this rotation is greater than each word from case (ii.a), this is the greatest rotation of w_k starting with $\mathtt{ba}$ and the thesis follows. □

Lemma 5.1.11 ($\beta(\mathtt{b}^j\mathtt{a})$ for all $2 \leq j \leq k-1$) Given the word $w_k = (\prod_{i=2}^{k-1} s_i e_i)q_k$ for some $k > 5$, and an integer $2 \leq i \leq k-2$, the first $k-i$ rotations in the BWT matrix starting with $\mathtt{b}^i\mathtt{a}$ are $\mathtt{b}^i\mathtt{aa}e_i \cdots \mathtt{a} < \mathtt{b}^i\mathtt{aa}e_{i+1} \cdots \mathtt{b} < \ldots < \mathtt{b}^i\mathtt{aa}e_{k-1} \cdots \mathtt{b} < \mathtt{b}^i\mathtt{a}s_2 \cdots \mathtt{b}$, followed by $\mathtt{b}^i\mathtt{aba}^{k-3}q_k \cdots \mathtt{b} < \mathtt{b}^i\mathtt{aba}^{k-4}s_{k-1} \cdots \mathtt{b} < \ldots < \mathtt{b}^i\mathtt{aba}^{i-1}s_{i+2} \cdots \mathtt{b} < \mathtt{b}^i\mathtt{aba}^{i-2}s_{i+1} \cdots \mathtt{a}$.

Proof. All runs of $\mathtt{b}$'s of length at least $2 \leq i \leq k-2$, either appear in (i) s_j or (ii) e_j, for all $i \leq j \leq k-1$, or in (iii) q_k. Let us consider the three cases separately. For all $i \leq j \leq k-1$, the rotation starting within s_j (i) has as prefix $\mathtt{b}^i\mathtt{aa}e_j$. For all $i \leq j \leq k-2$, the rotation starting within e_j (ii) has as prefix $\mathtt{b}^i\mathtt{aba}^{j-2}s_{j+1}$, and for $j = k-1$ we have the rotation with prefix $\mathtt{b}^i\mathtt{aba}^{k-3}q_k$. Finally, the rotation starting within q_k (iii) has as prefix $\mathtt{b}^i\mathtt{a}s_2$.

By construction, we can see that first we have all the rotations from case (i) sorted according to the lexicographic order of the words in $\bigcup_{j=i}^{k-1}\{e_i\}$ (Lemma 5.1.3), then we have the rotation from case (iii), and finally the rotation from case (ii), sorted according to the decreasing length of the run of $\mathtt{a}$'s following the common prefix $\mathtt{b}^i\mathtt{ab}$.

Moreover, note that only when the run of $\mathtt{b}$'s is of length exactly i the rotation end with an $\mathtt{a}$. Thus, the only for the rotations ending with an $\mathtt{a}$ are those starting within s_i and e_i, i.e., those with prefix $\mathtt{b}^i\mathtt{a}e_i$ and $\mathtt{b}^i\mathtt{aba}^{i-2}s_{i+1}$. □

Lemma 5.1.12 ($\beta(\mathtt{b}^k\mathtt{a})$) Given the word $w_k = (\prod_{i=2}^{k-1} s_i e_i)q_k$ for some $k > 5$, the last four rotations of the BWT matrix are $\mathtt{b}^{k-1}\mathtt{aa}e_{k-1} \cdots \mathtt{a} < \mathtt{b}^{k-1}\mathtt{a}s_2 \cdots \mathtt{b} < \mathtt{b}^{k-1}\mathtt{aba}^{k-3}q_k \cdots \mathtt{a} < \mathtt{b}^k\mathtt{a}s_2 \cdots \mathtt{a}$.

Proof. Observe that the only rotations with prefix $\mathtt{b}^{k-1}\mathtt{a}$ either start within s_{k-1}, or q_k, or e_{k-1}. These rotations have prefix respectively $\mathtt{b}^{k-1}\mathtt{aa}e_{k-1}$, $\mathtt{b}^{k-1}\mathtt{a}s_2$, and $\mathtt{b}^{k-1}\mathtt{aba}^{k-3}q_k$. One can see that these rotations taken in this order are already sorted, and only the rotation starting within q_k ends with a $\mathtt{b}$, while the other two with an $\mathtt{a}$. Finally, the only occurrence of $\mathtt{b}^k$ is within q_k. Hence, the last rotation in lexicographic order starts with $\mathtt{b}^k\mathtt{a}s_2$, and since the run of $\mathtt{b}$'s is maximal it ends with an $\mathtt{a}$, and the thesis follows. □

The following proposition puts together the BWT computations carried out for all blocks of consecutive rows, highlighting which prefixes are shared.

Block prefix	Ordering factor	BWT
$\mathtt{a}^{k-2}\mathtt{b}$	$\mathtt{b}^{k-1}\mathtt{a}$	b
$\mathtt{a}^{k-3}\mathtt{b}$	$\mathtt{b}^{k-2}\mathtt{aa}$	b
	$\mathtt{b}^{k-1}\mathtt{a}$	a
$\vdots$	$\vdots$	$\vdots$
$\mathtt{a}^{4}\mathtt{b}$	$\mathtt{b}^{5}\mathtt{aa}$	b
	$\mathtt{b}^{6}\mathtt{aa}$	a
	$\vdots$	$\vdots$
	$\mathtt{b}^{k-1}\mathtt{a}$	a
aaab	bab	b
	bbaba	b
	bbbabaa	b
	bbbbaa	b
	bbbbabaaa	b
	bbbbbaa	a
	bbbbbabaaaa	b
	$\vdots$	$\vdots$
	$\mathtt{b}^{k-2}\mathtt{aa}$	a
	$\mathtt{b}^{k-2}\mathtt{aba}^{k-3}$	b
	$\mathtt{b}^{k-1}\mathtt{a}$	a

Block prefix	Ordering factor	BWT
aab	baa	b
	bab	a
	bbaba	a
	bbbaa	b
	bbbabaa	b
	bbbbaa	a
	bbbbabaaa	a
	$\vdots$	$\vdots$
	$\mathtt{b}^{k-2}\mathtt{aa}$	a
	$\mathtt{b}^{k-2}\mathtt{aba}^{k-3}$	a
	$\mathtt{b}^{k-1}\mathtt{a}$	a
ab	$\mathtt{a}^{k-3}q_k$	b
	$\mathtt{a}^{k-4}s_{k-1}$	b
	$\vdots$	$\vdots$
	s_3	b
	baa	a
	bab	b
	bbaa	b
	bbaba	a
	bbbaa	a
	bbbabaa	a
	$\vdots$	$\vdots$
	$\mathtt{b}^{k-1}\mathtt{a}$	a

Block prefix	Ordering factor	BWT
ba	$\mathtt{a}^{k-4}q_k$	a
	$\mathtt{a}^{k-5}s_{k-1}$	a
	$\vdots$	$\vdots$
	$\mathtt{a}^{2}s_6$	a
	$\mathtt{a}e_2$	b
	$\mathtt{a}e_3$	b
	$\mathtt{a}e_4$	a
	$\mathtt{a}s_5$	a
	$\mathtt{a}e_5$	b
	$\mathtt{a}e_6$	b
	$\vdots$	$\vdots$
	$\mathtt{a}e_{k-1}$	b
	s_2	b
	s_4	a
	$\mathtt{ba}^{k-3}q_k$	b
	$\mathtt{ba}^{k-4}s_{k-1}$	b
	$\vdots$	$\vdots$
	$\mathtt{b}s_3$	b
	bbbaa	a

Block prefix	Ordering factor	BWT
bba	$\mathtt{a}e_2$	a
	$\mathtt{a}e_3$	b
	$\vdots$	$\vdots$
	$\mathtt{a}e_{k-1}$	b
	s_2	b
	$\mathtt{ba}^{k-3}q_k$	b
	$\mathtt{ba}^{k-4}s_{k-1}$	b
	$\vdots$	$\vdots$
	$\mathtt{ba}s_4$	b
	$\mathtt{ba}s_3$	a
$\vdots$	$\vdots$	$\vdots$
$\mathtt{b}^{k-1}\mathtt{a}$	$\mathtt{a}e_{k-1}$	a
	s_2	b
	$\mathtt{ba}^{k-3}q_k$	a
$\mathtt{b}^{k}\mathtt{a}$	s_2	a

Figure 5.1: Scheme of the BWT matrix of a word w_k with $k > 5$. The *block prefix* column shows the common prefix shared by all the rotations in a block. The *ordering factor* column shows the factor following the block prefix of a rotation, which decides its relative order inside its block. The BWT column shows the last character of each rotation. The dashed lines divide sub-ranges of rotations for which the BWT follows distinct patterns.

Proposition 5.1.13 Given an integer $k > 5$, let $w_k = (\prod_{i=2}^{k-1} s_i e_i) q_k$. Then,

$$\begin{aligned}
\beta(\mathtt{a}^i\mathtt{b}) &= \mathtt{ba}^{k-i-2} \text{ for all } 4 \le i \le k-2,\\
\beta(\mathtt{a}^3\mathtt{b}) &= \mathtt{b}^5(\mathtt{ab})^{k-6}\mathtt{a},\\
\beta(\mathtt{a}^2\mathtt{b}) &= \mathtt{baaba}^{2k-8},\\
\beta(\mathtt{ab}) &= \mathtt{b}^{k-2}\mathtt{aaba}^{2k-6},\\
\beta(\mathtt{ba}) &= \mathtt{a}^{k-5}\mathtt{bbbab}^{k-4}\mathtt{ab}^{k-2}\mathtt{a},\\
\beta(\mathtt{b}^j\mathtt{a}) &= \mathtt{ab}^{2k-2j-1}\mathtt{a} \text{ for all } 2 \le j \le k-1, \text{ and}\\
\beta(\mathtt{b}^k\mathtt{a}) &= \mathtt{a}.
\end{aligned}$$

Hence, the BWT of the w_k is $\mathtt{bwt}(w_k) = \prod_{i=2}^{k-1} \beta(\mathtt{a}^{k-i}\mathtt{b}) \cdot \prod_{i=1}^{k} \beta(\mathtt{b}^i\mathtt{a})$. Moreover, $r(w_k) = 6k - 12$.

Proof. The words $\beta(\mathtt{a}^{k-2}\mathtt{b})$, $\beta(\mathtt{a}^i\mathtt{b})$ for all $4 \le i \le k-2$, $\beta(\mathtt{a}^3\mathtt{b})$, $\beta(\mathtt{a}^2\mathtt{b})$, $\beta(\mathtt{ab})$, $\beta(\mathtt{ba})$, $\beta(\mathtt{b}^j\mathtt{a})$ for all $2 \le j \le k-1$, and $\beta(\mathtt{b}^k\mathtt{a})$, are the concatenations of the last characters of the rotations from Lemma 5.1.5, Lemma 5.1.6, Lemma 5.1.7, Lemma 5.1.8, Lemma 5.1.9, Lemma 5.1.10, Lemma 5.1.11, and Lemma 5.1.12 respectively. Moreover, every rotation used to build $\beta(\mathtt{a}^i\mathtt{b})$ is smaller than each rotation used to build $\beta(\mathtt{a}^{i'}\mathtt{b})$, for every $1 \le i' < i \le k-2$.

Symmetrically, every rotation used to build $\beta(\mathtt{b}^j\mathtt{a})$ is greater than each rotation used to build $\beta(\mathtt{b}^{j'}\mathtt{a})$, for every $1 \le j' < j \le k$. Since we have considered all the disjoint ranges of rotations of w_k based on their common prefix, the word $\prod_{i=2}^{k-1}\beta(\mathtt{a}^{k-i}\mathtt{b}) \cdot \prod_{i=1}^{k}\beta(\mathtt{b}^i\mathtt{a})$ is the BWT of w_k.

With the structure of $\mathtt{bwt}(w_k)$, we can easily derive its number of runs. The word $\prod_{i=2}^{k-4}(\beta(\mathtt{a}^{k-i}\mathtt{b}))$ has exactly $2(k-6)$ runs: we start with 2 runs from $\beta(\mathtt{a}^{k-2}\mathtt{b})\beta(\mathtt{a}^{k-3}\mathtt{b}) = \mathtt{bba}$, and then, concatenating each other $\beta(\mathtt{a}^i\mathtt{b})$ up to $\beta(\mathtt{a}^4\mathtt{b})$ adds 2 new runs each. It is easy to see that $\beta(\mathtt{aaab})$, $\beta(\mathtt{aab})$, and $\beta(\mathtt{ab})$, have $2(k-5)$, 4, and 4 runs, respectively. Moreover, the boundaries between these words do not merge, nor with $\beta(\mathtt{a}^4\mathtt{b})$ in the case of $\beta(\mathtt{aaab})$. The word $\beta(\mathtt{ba})$ has exactly 7 runs but it merges with $\beta(\mathtt{ab})$ and $\beta(\mathtt{bba})$, hence we only charge 5 runs to this word. The remaining part of the BWT, i.e., $\prod_{i=2}^{k}(\beta(\mathtt{b}^i\mathtt{a}))$, has $2(k-2)+1$ runs: we start with 3 runs from $\beta(\mathtt{bba})$, and then, concatenating each other $\beta(\mathtt{b}^i\mathtt{a})$ up to $\beta(\mathtt{b}^{k-1}\mathtt{a})$ adds 2 new runs each. The word $\beta(\mathtt{b}^k\mathtt{a})$ does not add new runs, as it consists only of an $\mathtt{a}$ that merges with the previous one. Overall, we have $2(k-6)+2(k-5)+4+4+5+2(k-2)+1 = 6k-12$, and the claim holds. □

5.1.2 BWT of w_k after an edit operation

The following lemmas describe the BWT of w_k after some specific edit operations are applied. Instead of proving the whole structure of the BWT from the beginning, we compare how the edit operation changes either the relative order or the last character of the rotations of w_k. To do so, in this part we use the notation $\beta(v)$ and $\beta^\star(v)$ to denote the BWT in correspondence of the rotations with prefix $v \in \Sigma^*$ of w_k and w'_k respectively, where w'_k is obtained after applying to w_k an specific edit operation. The number of runs in the BWT of w'_k can easily be derived by comparing its BWT to the BWT of w_k, for which we explicitly counted the number of runs, so we omit that part of the proofs. All the edit operations on w_k we show in this subsection increase the number $r(w_k)$ by a $\Theta(k)$ additive factor. To give an intuition, this increment comes mainly from the $\beta^\star(\mathtt{b}^j\mathtt{a})$ ranges for $2 \le j \le k-2$, because for each one of the corresponding ranges $\beta(\mathtt{b}^j\mathtt{a}) = \mathtt{ab}^{2k-2j-1}\mathtt{a}$ in $\mathtt{bwt}(w_k)$, one of the $\mathtt{b}$'s is either moved to the top or the bottom of the range, in a consistent manner for each j (it depends on the edit operation if the $\mathtt{b}$ goes to the top or the bottom of the range, but it is the same behavior for all the ranges considered). Then, two ranges that originally agreed on their last and first character in w_k are now separated by a $\mathtt{b}$, adding this way 2 new runs for each j.

Lemma 5.1.14 (BWT of $w_k\mathtt{a}$) Given an integer $k > 5$, for $w_k\mathtt{a}$ it holds that

$$\begin{aligned}
\beta^\star(\mathtt{a}^i\mathtt{b}) &= \mathtt{ba}^{k-i-2} \text{ for all } 4 \le i \le k-2,\\
\beta^\star(\mathtt{a}^3\mathtt{b}) &= \mathtt{bb}^5(\mathtt{ab})^{k-6}\mathtt{a},\\
\beta^\star(\mathtt{a}^2\mathtt{b}) &= \mathtt{aaaba}^{2k-8},\\
\beta^\star(\mathtt{ab}) &= \mathtt{b}^{k-2}\mathtt{aaba}^{2k-6},\\
\beta^\star(\mathtt{ba}) &= \mathtt{a}^{k-5}\mathtt{bbbbab}^{k-5}\mathtt{ab}^{k-2}\mathtt{a},\\
\beta^\star(\mathtt{b}^j\mathtt{a}) &= \mathtt{bab}^{2k-2j-2}\mathtt{a} \text{ for all } 2 \le j \le k-1 \text{ and}\\
\beta^\star(\mathtt{b}^k\mathtt{a}) &= \mathtt{a}.
\end{aligned}$$

Hence, $\mathtt{bwt}(w_k\mathtt{a}) = \prod_{i=2}^{k-1}\beta^\star(\mathtt{a}^{k-i}\mathtt{b}) \cdot \prod_{i=1}^{k}\beta^\star(\mathtt{b}^i\mathtt{a})$. Moreover, it holds that $r(w_k\mathtt{a}) = 8k-20$.

Proof. By Lemmas 5.1.5 and 5.1.6, we can see that appending an $\mathtt{a}$ after q_k does not affect the BWT in the range of rotations having $\mathtt{a}^i\mathtt{b}$ as prefix, for all $4 \leq i \leq k-2$. Thus, $\beta^\star(\mathtt{a}^i\mathtt{b}) = \beta(\mathtt{a}^i\mathtt{b})$ for all $4 \leq i \leq k-2$.

The rotation starting with $\mathtt{aa}s_2$, which is not a circular factor of w_k, ends with a $\mathtt{b}$. By Lemma 5.1.7, we can see that such a rotation is the smallest one with prefix $\mathtt{aaab}$ in lexicographic order, while the other rotations maintain their relative order. Therefore, $\beta^\star(\mathtt{aaab}) = \mathtt{b} \cdot \beta(\mathtt{aaab})$.

By Lemma 5.1.8, the rotation with prefix $\mathtt{a}s_2$ is still the smallest rotation starting with $\mathtt{aab}$, with the difference that in this case, it ends with the last $\mathtt{a}$ of q_k. It follows that $\beta^\star(\mathtt{aab})$ is obtained by replacing the first $\mathtt{b}$ of $\beta(\mathtt{aab})$ with an $\mathtt{a}$.

Both the order and the last symbol of all the rotations having as prefix $\mathtt{ab}$ described in Lemma 5.1.9 is not affected from the insertion of the $\mathtt{a}$, and therefore $\beta^\star(\mathtt{ab}) = \beta(\mathtt{ab})$.

Let us now consider all the rotations of w_k with prefix $\mathtt{b}^j\mathtt{a}s_2$, for all $1 \leq j \leq k$. One can notice that $w_k\mathtt{a}$ does not have any rotation starting with $\mathtt{b}^j\mathtt{a}s_2$, for all $1 \leq j \leq k$, but instead it has rotations starting with $\mathtt{b}^j\mathtt{aa}s_2$. Thus, for all $1 \leq j \leq k-1$, to obtain $\beta^\star(\mathtt{b}^j\mathtt{a})$ from $\beta(\mathtt{b}^j\mathtt{a})$ we have to remove the $\mathtt{b}$ in correspondence of the rotations starting with $\mathtt{b}^j\mathtt{a}s_2$, and add a $\mathtt{b}$ in correspondence of the rotations $\mathtt{b}^j\mathtt{aa}s_2$. By Lemmas 5.1.10, 5.1.11, and 5.1.12, such rotations are placed right before the rotation starting with $\mathtt{b}^j\mathtt{aa}e_2$.

Finally, the last rotation has still the same prefix $\mathtt{b}^k\mathtt{a}$ and ends with an $\mathtt{a}$, and the thesis follows. □

Lemma 5.1.15 (BWT of $\widehat{w_k}$) Given an integer $k > 5$, for $\widehat{w_k}$ it holds that

$$\begin{aligned}
\beta^\star(\mathtt{a}^i\mathtt{b}) &= \mathtt{ba}^{k-i-2} \text{ for all } 4 \leq i \leq k-2,\\
\beta^\star(\mathtt{a}^3\mathtt{b}) &= \mathtt{b}^5(\mathtt{ab})^{k-6}\mathtt{a},\\
\beta^\star(\mathtt{a}^2\mathtt{b}) &= \mathtt{aaba}^{2k-8},\\
\beta^\star(\mathtt{ab}) &= \mathtt{b}^{k-2}\mathtt{baba}^{2k-6},\\
\beta^\star(\mathtt{ba}) &= \mathtt{a}^{k-5}\mathtt{bbbab}^{k-5}\mathtt{ab}^{k-2}\mathtt{ba},\\
\beta^\star(\mathtt{b}^j\mathtt{a}) &= \mathtt{ab}^{2k-2j-2}\mathtt{ab} \text{ for all } 2 \leq j \leq k-1 \text{ and}\\
\beta^\star(\mathtt{b}^k\mathtt{a}) &= \mathtt{a}.
\end{aligned}$$

Hence, $\mathtt{bwt}(\widehat{w_k}) = \prod_{i=2}^{k-1} \beta^\star(\mathtt{a}^{k-i}\mathtt{b}) \cdot \prod_{i=1}^{k} \beta^\star(\mathtt{b}^i\mathtt{a})$. Moreover, it holds that $r(\widehat{w_k}) = 8k-20$.

Proof. Analogously to the previous Lemma, if we look in Lemmas 5.1.5, 5.1.6, and 5.1.7, at the structure of the BWT in correspondence of the rotations starting with $\mathtt{a}^i\mathtt{b}$, for all $3 \leq i \leq k-2$, we can notice that the order or the symbols in the BWT is not affected. Thus, for all $3 \leq i \leq k-2$, we have $\beta^\star(\mathtt{a}^i\mathtt{b}) = \beta(\mathtt{a}^i\mathtt{b})$.

Since the last $\mathtt{a}$ of q_k is omitted, the circular factor $\mathtt{a}s_2$ does not appear anymore in $\widehat{w}$. Thus, $\beta^\star(\mathtt{aab})$ is obtained by removing the first $\mathtt{b}$ from $\beta(\mathtt{aab})$, since by Lemma 5.1.8 it is in correspondence of the rotation with prefix $\mathtt{a}s_2$.

On the other hand, we can observe from Lemma 5.1.9 that the rotation with prefix s_2 maintains its relative order also in $\widehat{w_k}$, but its last symbol is now a $\mathtt{b}$ instead of an $\mathtt{a}$.

For each $1 \leq j \leq k$, the rotation starting with $\mathtt{b}^j\mathtt{a}s_2$ of w_k does not appear in $\widehat{w_k}$, but in fact it is replaced by one having $\mathtt{b}^j s_2$ as prefix and ending in the same way. When $j = 1$, by Lemma 5.1.10 such a rotation is located between the last two rotations with the prefix $\mathtt{ba}$, which start by $\mathtt{bab}s_3$ and $\mathtt{b}s_3$ respectively. When $2 \leq j \leq k-1$, by Lemmas 5.1.11 and 5.1.12, the rotation starting with $\mathtt{b}^j s_2$ is greater than all the other rotations with prefix $\mathtt{b}^j\mathtt{a}$. Thus, for all $1 \leq j \leq k-1$, we obtain $\beta^\star(\mathtt{b}^j\mathtt{a})$ by moving the $\mathtt{b}$ in correspondence of the rotation starting with $\mathtt{ba}s_2$ from $\beta(\mathtt{b}^j\mathtt{a})$ and placing it in correspondence of $\mathtt{b}^j s_2$. Finally, the last rotation has still the same prefix $\mathtt{b}^k\mathtt{a}$ and ends with an $\mathtt{a}$, and the thesis follows. □

Lemma 5.1.16 (BWT of $\widehat{w_k}\mathtt{b}$) Given an integer $k > 5$, for $\widehat{w_k}\mathtt{b}$ it holds that

$$\begin{aligned}
\beta^\star(\mathtt{a}^i\mathtt{b}) &= \mathtt{ba}^{k-i-2} \text{ for all } 4 \leq i \leq k-2,\\
\beta^\star(\mathtt{a}^3\mathtt{b}) &= \mathtt{b}^5(\mathtt{ab})^{k-6}\mathtt{a},\\
\beta^\star(\mathtt{a}^2\mathtt{b}) &= \mathtt{aaba}^{2k-8},\\
\beta^\star(\mathtt{ab}) &= \mathtt{b}^{k-2}\mathtt{baba}^{2k-6},\\
\beta^\star(\mathtt{ba}) &= \mathtt{a}^{k-5}\mathtt{bbbab}^{k-5}\mathtt{ab}^{k-2}\mathtt{ba},\\
\beta^\star(\mathtt{b}^j\mathtt{a}) &= \mathtt{ab}^{2k-2j-2}\mathtt{ab} \text{ for all } 2 \leq j \leq k-1,\\
\beta^\star(\mathtt{b}^k\mathtt{a}) &= \mathtt{b} \text{ and}\\
\beta^\star(\mathtt{b}^{k+1}\mathtt{a}) &= \mathtt{a}.
\end{aligned}$$

Hence, $\mathtt{bwt}(\widehat{w_k}\mathtt{b}) = \prod_{i=2}^{k-1} \beta^\star(\mathtt{a}^{k-i}\mathtt{b}) \cdot \prod_{i=1}^{k+1} \beta^\star(\mathtt{b}^i\mathtt{a})$. Moreover, it holds that $r(\widehat{w_k}\mathtt{b}) = 8k - 20$.

Proof. For the rotations in correspondence of the rotations starting with an $\mathtt{a}$, notice that replacing the last $\mathtt{a}$ of w_k for a $\mathtt{b}$ or removing the last $\mathtt{a}$ affects the BWT in the same way. Therefore, $\beta^\star(\mathtt{a}^i\mathtt{b})$ is the same as Lemma 5.1.15 for all $1 \leq i \leq k-2$.

The same behaviour can be noticed on the rotations with prefix $\mathtt{b}^j\mathtt{a}$, for all $1 \leq j \leq k-1$, while the rotation starting with $\mathtt{b}^k\mathtt{a}$ is now preceded by a $\mathtt{b}$.

With respect to the other edit operations, we have the range of rotations starting with $\mathtt{b}^{k+1}\mathtt{a}$, which consists solely in $\mathtt{b}^{k+1}s_2 \cdots \mathtt{a}$. □

The structure of the BWT of w_k and other words obtained by applying one or more edit operations on w_k are summed up in Figure 5.2.

For a given word $w \neq \varepsilon$, let w^{ins}, w^{del}, and w^{sub} be the words obtained by applying on w an insertion, a deletion, and a substitution of a character respectively.

We compare the number of runs of w_k and its variations and notice that the difference after applying one of the edit operations is $\Theta(k)$ in the three cases.

Word	$\beta(\$)$	$\beta(\mathtt{a}\$)$	$\beta(\mathtt{aa}\$)$	$\beta(\mathtt{a}^i\mathtt{b})$	$\beta(\mathtt{a}^3\mathtt{b})$	$\beta(\mathtt{a}^2\mathtt{b})$	$\beta(\mathtt{ab})$
w_k	ε	ε	ε	$\mathtt{ba}^{k-i-2}$	$\mathtt{b}^5(\mathtt{ab})^{k-6}\mathtt{a}$	$\mathtt{baaba}^{2k-8}$	$\mathtt{b}^{k-2}\mathtt{aaba}^{2k-6}$
$w_k\mathtt{a}$	ε	ε	ε	$\mathtt{ba}^{k-i-2}$	$\mathtt{bb}^5(\mathtt{ab})^{k-6}\mathtt{a}$	$\mathtt{aaaba}^{2k-8}$	$\mathtt{b}^{k-2}\mathtt{aaba}^{2k-6}$
$\widehat{w_k}$	ε	ε	ε	$\mathtt{ba}^{k-i-2}$	$\mathtt{b}^5(\mathtt{ab})^{k-6}\mathtt{a}$	$\mathtt{aaba}^{2k-8}$	$\mathtt{b}^{k-2}\mathtt{baba}^{2k-6}$
$\widehat{w_k}\mathtt{b}$	ε	ε	ε	$\mathtt{ba}^{k-i-2}$	$\mathtt{b}^5(\mathtt{ab})^{k-6}\mathtt{a}$	$\mathtt{aaba}^{2k-8}$	$\mathtt{b}^{k-2}\mathtt{baba}^{2k-6}$
$w_k\$$	$\mathtt{a}$	$\mathtt{b}$	ε	$\mathtt{ba}^{k-i-2}$	$\mathtt{b}^5(\mathtt{ab})^{k-6}\mathtt{a}$	$\mathtt{aaba}^{2k-8}$	$\mathtt{b}^{k-2}\$\mathtt{aba}^{2k-6}$
$w_k\mathtt{b}\$$	$\mathtt{b}$	ε	ε	$\mathtt{ba}^{k-i-2}$	$\mathtt{b}^5(\mathtt{ab})^{k-6}\mathtt{a}$	$\mathtt{aaba}^{2k-8}$	$\mathtt{bb}^{k-2}\$\mathtt{aba}^{2k-6}$
$w_k\mathtt{bb}\$$	$\mathtt{b}$	ε	ε	$\mathtt{ba}^{k-i-2}$	$\mathtt{b}^5(\mathtt{ab})^{k-6}\mathtt{a}$	$\mathtt{aaba}^{2k-8}$	$\mathtt{bb}^{k-2}\$\mathtt{aba}^{2k-6}$
$w_k\mathtt{a}\$$	$\mathtt{a}$	$\mathtt{a}$	$\mathtt{b}$	$\mathtt{ba}^{k-i-2}$	$\mathtt{b}^5(\mathtt{ab})^{k-6}\mathtt{a}$	$\mathtt{aaba}^{2k-8}$	$\mathtt{b}^{k-2}\$\mathtt{aba}^{2k-6}$

Word	$\beta(\mathtt{b}\$)$	$\beta(\mathtt{ba})$	$\beta(\mathtt{bb}\$)$	$\beta(\mathtt{b}^j\mathtt{a})$	$\beta(\mathtt{b}^k\mathtt{a})$	$\beta(\mathtt{b}^{k+1})$	$r(\cdot)$
w_k	ε	$\mathtt{a}^{k-5}\mathtt{bbbab}^{k-4}\mathtt{ab}^{k-2}\mathtt{a}$	ε	$\mathtt{ab}^{2k-2j-1}\mathtt{a}$	$\mathtt{a}$	ε	$6k-12$
$w_k\mathtt{a}$	ε	$\mathtt{a}^{k-5}\mathtt{bbbbab}^{k-5}\mathtt{ab}^{k-2}\mathtt{a}$	ε	$\mathtt{bab}^{2k-2j-2}\mathtt{a}$	$\mathtt{a}$	ε	$8k-20$
$\widehat{w_k}$	ε	$\mathtt{a}^{k-5}\mathtt{bbbab}^{k-5}\mathtt{ab}^{k-2}\mathtt{ba}$	ε	$\mathtt{ab}^{2k-2j-2}\mathtt{ab}$	$\mathtt{a}$	ε	$8k-20$
$\widehat{w_k}\mathtt{b}$	ε	$\mathtt{a}^{k-5}\mathtt{bbbab}^{k-5}\mathtt{ab}^{k-2}\mathtt{ba}$	ε	$\mathtt{ab}^{2k-2j-2}\mathtt{ab}$	$\mathtt{b}$	$\mathtt{a}$	$8k-20$
$w_k\$$	ε	$\mathtt{ba}^{k-5}\mathtt{bbbab}^{k-5}\mathtt{ab}^{k-2}\mathtt{a}$	ε	$\mathtt{bab}^{2k-2j-2}\mathtt{a}$	$\mathtt{a}$	ε	$8k-16$
$w_k\mathtt{b}\$$	$\mathtt{a}$	$\mathtt{a}^{k-5}\mathtt{bbbab}^{k-5}\mathtt{abb}^{k-2}\mathtt{a}$	ε	$\mathtt{ab}^{2k-2j-1}\mathtt{a}$	$\mathtt{a}$	ε	$6k-13$
$w_k\mathtt{bb}\$$	$\mathtt{b}$	$\mathtt{a}^{k-5}\mathtt{bbbab}^{k-5}\mathtt{abb}^{k-2}\mathtt{a}$	$\mathtt{a}$	$\mathtt{ab}^{2k-2j-2}\mathtt{ab}$	$\mathtt{a}$	ε	$8k-17$
$w_k\mathtt{a}\$$	ε	$\mathtt{ba}^{k-5}\mathtt{bbbab}^{k-5}\mathtt{ab}^{k-2}\mathtt{a}$	ε	$\mathtt{bab}^{2k-2j-2}\mathtt{a}$	$\mathtt{a}$	ε	$8k-16$

Figure 5.2: BWTs of the word w_k and its variants after different edit operations. The word in the intersection of the column $\beta(x)$ with the row w is the range of $\mathtt{bwt}(w)$ corresponding to all the rotations that have x as a prefix. The columns $\beta(\mathtt{a}^i\mathtt{b})$ and $\beta(\mathtt{b}^j\mathtt{a})$ represent ranges of columns from $i \in [k-2\,..\,4]$ (in that order) and $j \in [2\,..\,k-1]$, respectively. Note that the prefixes in the columns are disjoint, and cover all the possible ranges for the set of words considered. The BWT of each word is the concatenation of all the words in its row from left to right. In the last column appears the number of BWT runs of each of these words.

Proposition 5.1.17 There exists an infinite family of words w such that: (i) $r(w^{ins})-r(w) = \Theta(\sqrt{n})$; (ii) $r(w^{del}) - r(w) = \Theta(\sqrt{n})$; (iii) $r(w^{sub}) - r(w) = \Theta(\sqrt{n})$.

Proof. The family is composed of the words w_k with $k > 5$. Let $n = |w_k|$. If $w_k^{ins} = w_k\mathtt{a}$, $w_k^{del} = \widehat{w_k}$, and $w_k^{sub} = \widehat{w_k}\mathtt{b}$, from Proposition 5.1.13, Lemma 5.1.14, Lemma 5.1.15, and Lemma 5.1.16, we have that $r(w_k\mathtt{a}) = r(\widehat{w_k}) = r(\widehat{w_k}\mathtt{b}) = r(w_k) + (2k-8)$. From Observation 5.1.2, we have that $2k-8 = \Theta(\sqrt{n})$. □

5.1.3 Additive sensitivity for $r_\$$

In this subsection, we discuss the additive sensitivity when $r_\$$ is considered. Similar results have been proven regarding the multiplicative sensitivity [60].

The following proposition [60] shows that for some specific edit operations $r_\$$ might not be majorly affected. This result will be used later on some propositions.

Proposition 5.1.18 ([60]) Let c be smaller than or equal to the smallest character in a word v, then $r_{\$}(v) \leq r_{\$}(vc) \leq r_{\$}(v) + 1$.

In general, appending, deleting, or substituting with a symbol that is not the smallest of the alphabet can increase the number of runs of a word by an additive factor of $\Theta(\sqrt{n})$. We show this in the following series of lemmas.

Lemma 5.1.19 (BWT of $w_k\$$) Given an integer $k > 5$, for $w_k\$$ it holds that

$$\begin{aligned}
\beta^\star(\$) &= \mathtt{a} \\
\beta^\star(\mathtt{a\$}) &= \mathtt{b} \\
\beta^\star(\mathtt{a}^i\mathtt{b}) &= \mathtt{ba}^{k-i-2} \text{ for all } 4 \leq i \leq k-2, \\
\beta^\star(\mathtt{a}^3\mathtt{b}) &= \mathtt{b}^5(\mathtt{ab})^{k-6}\mathtt{a}, \\
\beta^\star(\mathtt{a}^2\mathtt{b}) &= \mathtt{aaba}^{2k-8}, \\
\beta^\star(\mathtt{ab}) &= \mathtt{b}^{k-2}\$\mathtt{aba}^{2k-6}, \\
\beta^\star(\mathtt{ba}) &= \mathtt{ba}^{k-5}\mathtt{bbbab}^{k-5}\mathtt{ab}^{k-2}\mathtt{a}, \\
\beta^\star(\mathtt{b}^j\mathtt{a}) &= \mathtt{bab}^{2k-2j-2}\mathtt{a} \text{ for all } 2 \leq j \leq k-1 \text{ and} \\
\beta^\star(\mathtt{b}^k\mathtt{a}) &= \mathtt{a}.
\end{aligned}$$

Hence, $\mathtt{bwt}(w_k\$) = \beta^\star(\$) \cdot \beta^\star(\mathtt{a\$}) \cdot \prod_{i=2}^{k-1} \beta^\star(\mathtt{a}^{k-i}\mathtt{b}) \cdot \prod_{i=1}^{k} \beta^\star(\mathtt{b}^i\mathtt{a})$. Moreover, it holds that $r(w_k\$) = 8k - 16$.

Proof. The first rotation of $\mathtt{bwt}(w_k\$)$ is $\$w_k$ and ends with an $\mathtt{a}$ because w_k ends with an $\mathtt{a}$. Hence, $\beta^\star(\$) = \mathtt{a}$. There is also a rotation $\mathtt{a}\$\widehat{w_k}$, which ends with a $\mathtt{b}$ because $\widehat{w_k}$ ends with a $\mathtt{b}$. Hence, $\beta^\star(\mathtt{a\$}) = \mathtt{b}$. It is left to compare the remaining ranges $\beta^\star(v)$ with respect to $\beta(v)$.

It is easy to see from Lemma 5.1.5, Lemma 5.1.6, and Lemma 5.1.7 that $\beta^\star(\mathtt{a}^i\mathtt{b}) = \beta(\mathtt{a}^i\mathtt{b})$ for all $3 \leq i \leq k-2$.

The rotation starting with $\mathtt{a}s_2$ in w_k does not exist anymore when $\$$ is appended to w_k. By Lemma 5.1.8 the remaining rotations keep their last symbols and relative order. Therefore, $\beta^\star(\mathtt{aab})$ is the same as $\beta(\mathtt{aab})$ but with the first character removed, i.e., $\beta^\star(\mathtt{aab}) = \mathtt{aaba}^{2k-8}$.

For the rotations starting with $\mathtt{ab}$, it happens that the rotation that originally started with s_2 in w_k, now ends with a $\$$. By Lemma 5.1.9, the remaining rotations do not change their last symbol. Also, all the rotations keep their relative order. Hence, $\beta^\star(\mathtt{ab}) = \mathtt{b}^{k-2}\$\mathtt{aba}^{2k-6}$.

In the case of the rotations starting with $\mathtt{ba}$, the rotation that originally started with $\mathtt{ba}s_2$ now starts with $\mathtt{ba}\$s_2$ and is the smallest of its range. From Lemma 5.1.10 the remaining rotations keep their last symbols and relative order. Hence, $\beta^\star(\mathtt{ba}) = \mathtt{ba}^{k-5}\mathtt{bbbab}^{k-5}\mathtt{ab}^{k-2}\mathtt{a}$.

For the rotations starting with $\mathtt{b}^j\mathtt{a}$ for $2 \leq j \leq k-1$, one can notice that after appending $\$$ to w_k, the rotation that previously started with $\mathtt{b}^j\mathtt{a}s_2$ and ended with a $\mathtt{b}$, now starts with $\mathtt{b}^j\mathtt{a}\$s_2$ and still ends with a $\mathtt{b}$. Moreover, this rotation is smaller than the rotation starting with $\mathtt{b}^j\mathtt{aa}e_j$. From Lemma 5.1.11 and Lemma 5.1.12 we can see that all the other rotations keep their relative order and last symbols. The rotation starting with $\mathtt{b}^j\mathtt{aa}e_j$ still ends with an $\mathtt{a}$, but now is the second smallest of its range. Hence, $\beta^\star(\mathtt{b}^j\mathtt{a}) = \mathtt{bab}^{2k-2j-2}\mathtt{a}$ for all $2 \leq j \leq k-1$.

Finally, it is clear that $\beta^\star(\mathtt{b}^k\mathtt{a}) = \mathtt{a}$, as there is only one maximal run of k symbol $\mathtt{b}$'s, and it is not preceded by \$. □

Lemma 5.1.20 (BWT of $w_k\mathtt{b}\$$) Given an integer $k > 5$, for $w_k\mathtt{b}\$$ it holds that

$$\begin{aligned}
\beta^\star(\$) &= \mathtt{b} \\
\beta^\star(\mathtt{a}^i\mathtt{b}) &= \mathtt{ba}^{k-i-2} \text{ for all } 4 \leq i \leq k-2, \\
\beta^\star(\mathtt{a}^3\mathtt{b}) &= \mathtt{b}^5(\mathtt{ab})^{k-6}\mathtt{a}, \\
\beta^\star(\mathtt{a}^2\mathtt{b}) &= \mathtt{aaba}^{2k-8}, \\
\beta^\star(\mathtt{ab}) &= \mathtt{bb}^{k-2}\$\mathtt{aba}^{2k-6}, \\
\beta^\star(\mathtt{b}\$) &= \mathtt{a}, \\
\beta^\star(\mathtt{ba}) &= \mathtt{a}^{k-5}\mathtt{bbbab}^{k-5}\mathtt{abb}^{k-2}\mathtt{a}, \\
\beta^\star(\mathtt{b}^j\mathtt{a}) &= \mathtt{ab}^{2k-2j-1}\mathtt{a} \text{ for all } 2 \leq j \leq k-1 \text{ and} \\
\beta^\star(\mathtt{b}^k\mathtt{a}) &= \mathtt{a}.
\end{aligned}$$

Hence, $\mathtt{bwt}(w_k\mathtt{b}\$) = \beta^\star(\$) \cdot (\prod_{i=2}^{k-1} \beta^\star(\mathtt{a}^{k-i}\mathtt{b})) \cdot \beta^\star(\mathtt{b}\$) \cdot (\prod_{i=1}^{k} \beta^\star(\mathtt{b}^i\mathtt{a}))$. Moreover, it holds that $r(w_k\mathtt{b}\$) = 6k - 13$.

Proof. The first rotation of $\mathtt{bwt}(w_k\mathtt{b}\$)$ is $\$w_k\mathtt{b}$. Hence, $\beta^\star(\$) = \mathtt{b}$. There is also a rotation $\mathtt{b}\$w_k$, which ends with an $\mathtt{a}$ because w_k ends with an $\mathtt{a}$. Hence, $\beta^\star(\mathtt{b}\$) = \mathtt{a}$. It is left to compare the remaining ranges $\beta^\star(v)$ with respect to $\beta(v)$.

It is easy to see from Lemma 5.1.5, Lemma 5.1.6, and Lemma 5.1.7 that $\beta^\star(\mathtt{a}^i\mathtt{b}) = \beta(\mathtt{a}^i\mathtt{b})$ for all $3 \leq i \leq k-2$.

The rotation starting with $\mathtt{a}s_2$ in w_k does not exist anymore when $\mathtt{b}\$$ is appended to w_k. By Lemma 5.1.8 the remaining rotations keep their last symbols and relative order. Therefore, $\beta^\star(\mathtt{aab})$ is the same as $\beta(\mathtt{aab})$ but with the first character removed, i.e., $\beta^\star(\mathtt{aab}) = \mathtt{aaba}^{2k-8}$.

For the rotations starting with $\mathtt{ab}$, it happens that the rotation that originally started with s_2 in w_k, now ends with a \$ when $\mathtt{b}\$$ is appended. Also, there is a new rotation starting with $\mathtt{ab}\$$ that ends with $\mathtt{b}$, and is clearly the smallest of the range. By Lemma 5.1.9, the remaining rotations do not change their last symbol. Also, all the rotations that come from w_k keep their relative order. Hence, $\beta^\star(\mathtt{ab}) = \mathtt{bb}^{k-2}\$\mathtt{aba}^{2k-6}$.

In the case of the rotations starting with $\mathtt{ba}$, the rotation that originally started with $\mathtt{ba}s_2$ now starts with $\mathtt{bab}\$s_2$ and can be found just before the rotation starting with $\mathtt{baba}^{k-2}$. From Lemma 5.1.10 the remaining rotations keep their last symbols and relative order. Hence, $\beta^\star(\mathtt{ba}) = \mathtt{a}^{k-5}\mathtt{bbbab}^{k-5}\mathtt{abb}^{k-2}\mathtt{a}$.

For the rotations starting with $\mathtt{b}^j\mathtt{a}$ for $2 \leq j \leq k-1$, one can notice that after appending $\mathtt{b}\$$ to w_k, the rotation that previously started with $\mathtt{b}^j\mathtt{a}s_2$ and ended with a $\mathtt{b}$, now starts with $\mathtt{b}^j\mathtt{ab}\$s_2$ and still ends with a $\mathtt{b}$. Moreover, this rotation is still strictly in between the rotations starting with $\mathtt{b}^j\mathtt{aa}e_j$ and $\mathtt{b}^j\mathtt{aba}^{j-2}s_{j+1}$ (q_k instead of s_{j+1} if $j = k-1$). From Lemma 5.1.11 and Lemma 5.1.12, we can see that the latter two rotations are still the smallest and greatest of the range, and both end with an $\mathtt{a}$. Also, all the other rotations keep their last symbols. Hence, $\beta^\star(\mathtt{b}^j\mathtt{a}) = \beta(\mathtt{b}^j\mathtt{a})$ for all $2 \le j \le k-1$.

Finally, it is clear that $\beta^\star(\mathtt{b}^k\mathtt{a}) = \mathtt{a}$, as there is only one maximal run of k symbol $\mathtt{b}$'s, and it is not preceded by \$. □

Lemma 5.1.21 (BWT of $w_k\mathtt{bb}\$$) Given an integer $k > 5$, for $w_k\mathtt{bb}\$$ it holds that

$$\begin{aligned}
\beta^\star(\$) &= \mathtt{b} \\
\beta^\star(\mathtt{a}^i\mathtt{b}) &= \mathtt{ba}^{k-i-2} \text{ for all } 4 \le i \le k-2, \\
\beta^\star(\mathtt{a}^3\mathtt{b}) &= \mathtt{b}^5(\mathtt{ab})^{k-6}\mathtt{a}, \\
\beta^\star(\mathtt{a}^2\mathtt{b}) &= \mathtt{aaba}^{2k-8}, \\
\beta^\star(\mathtt{ab}) &= \mathtt{bb}^{k-2}\$\mathtt{aba}^{2k-6}, \\
\beta^\star(\mathtt{b}\$) &= \mathtt{b}, \\
\beta^\star(\mathtt{ba}) &= \mathtt{a}^{k-5}\mathtt{bbbab}^{k-5}\mathtt{abb}^{k-2}\mathtt{a}, \\
\beta^\star(\mathtt{bb}\$) &= \mathtt{a}, \\
\beta^\star(\mathtt{b}^j\mathtt{a}) &= \mathtt{ab}^{2k-2j-2}\mathtt{ab} \text{ for all } 2 \le j \le k-1 \text{ and} \\
\beta^\star(\mathtt{b}^k\mathtt{a}) &= \mathtt{a}.
\end{aligned}$$

Hence, $\mathtt{bwt}(w_k\mathtt{bb}\$) = \beta^\star(\$) \cdot (\prod_{i=2}^{k-1} \beta^\star(\mathtt{a}^{k-i}\mathtt{b})) \cdot \beta^\star(\mathtt{b}\$) \cdot \beta^\star(\mathtt{ba}) \cdot \beta^\star(\mathtt{bb}\$) \cdot (\prod_{i=2}^{k} \beta^\star(\mathtt{b}^i\mathtt{a}))$. Moreover, it holds that $r(w_k\mathtt{b}\$) = 8k - 17$.

Proof. The first rotation of $\mathtt{bwt}(w_k\mathtt{bb}\$)$ is $\$w_k\mathtt{bb}$. Hence, $\beta^\star(\$) = \mathtt{b}$. There is another new rotation $\mathtt{b}\$w_k\mathtt{b}$. Hence, $\beta^\star(\mathtt{b}\$) = \mathtt{b}$. There is also a rotation $\mathtt{bb}\$w_k$ that ends with an $\mathtt{a}$ because w_k ends with an $\mathtt{a}$. Hence, $\beta^\star(\mathtt{bb}\$) = \mathtt{a}$. It is left to compare the remaining ranges $\beta^\star(v)$ with respect to $\beta(v)$.

It is easy to see from Lemma 5.1.5, Lemma 5.1.6, and Lemma 5.1.7 that $\beta^\star(\mathtt{a}^i\mathtt{b}) = \beta(\mathtt{a}^i\mathtt{b})$ for all $3 \le i \le k-2$.

The rotation starting with $\mathtt{a}s_2$ in w_k does not exist anymore when $\mathtt{bb}\$$ is appended to w_k. By Lemma 5.1.8 the remaining rotations keep their last symbols and relative order. Therefore, $\beta^\star(\mathtt{aab})$ is the same as $\beta(\mathtt{aab})$ but with the first character removed, i.e., $\beta^\star(\mathtt{aab}) = \mathtt{aaba}^{2k-8}$.

For the rotations starting with $\mathtt{ab}$, it happens that the rotation that originally started with s_2 in w_k, now ends with a \$ when $\mathtt{bb}\$$ is appended. Also, there is a new rotation starting with $\mathtt{abb}\$$ that ends with $\mathtt{b}$, and can be found just before the rotation starting with s_2. By Lemma 5.1.9, the remaining rotations do not change their last symbol. Also, all the rotations that come from w_k keep their relative order. Hence, $\beta^\star(\mathtt{ab}) = \mathtt{b}^{k-2}\mathtt{b}\$\mathtt{aba}^{2k-6}$.

In the case of the rotations starting with $\mathtt{ba}$, the rotation that originally started with $\mathtt{ba}s_2$ now starts with $\mathtt{babb}\$s_2$ and can be found just before the rotation starting with $\mathtt{b}s_3$ (the greatest on the range). From Lemma 5.1.10 we can see that the remaining rotations keep their last symbols and relative order. Hence, $\beta^\star(\mathtt{ba}) = \mathtt{a}^{k-5}\mathtt{bbbab}^{k-5}\mathtt{ab}^{k-2}\mathtt{ba}$.

For the rotations starting with $\mathtt{b}^j\mathtt{a}$ for $2 \leq j \leq k-1$, one can notice that after appending $\mathtt{bb}\$$ to w_k, the rotation that previously started with $\mathtt{b}^j\mathtt{a}s_2$ and ended with a $\mathtt{b}$, now starts with $\mathtt{b}^j\mathtt{abb}\$s_2$ and still ends with a $\mathtt{b}$. Moreover, this rotation is greater than the rotation starting with $\mathtt{b}^j\mathtt{aba}^{j-2}s_{j+1}$ (q_k instead of s_{j+1} if $j = k-1$). From Lemma 5.1.11 and Lemma 5.1.12 we can see that all the other rotations keep their relative order an last symbols. The rotation starting with $\mathtt{b}^j\mathtt{aba}^{j-2}s_{j+1}$ (q_k instead of s_{j+1} if $j = k-1$) still ends with an $\mathtt{a}$, but now is the second greatest of its range. Hence, $\beta^\star(\mathtt{b}^j\mathtt{a}) = \mathtt{ab}^{2k-2j-2}\mathtt{ab}$ for all $2 \leq j \leq k-1$.

Finally, it is clear that $\beta^\star(\mathtt{b}^k\mathtt{a}) = \mathtt{a}$, as there is only one maximal run of k symbol $\mathtt{b}$'s, and it is not preceded by \$. □

Lemma 5.1.22 (BWT of $w_k\mathtt{a}\$$) Given an integer $k > 5$, for $w_k\mathtt{a}\$$ it holds that

$$\begin{aligned}
\beta^\star(\$) &= \mathtt{a}\\
\beta^\star(\mathtt{a}\$) &= \mathtt{a}\\
\beta^\star(\mathtt{aa}\$) &= \mathtt{b}\\
\beta^\star(\mathtt{a}^i\mathtt{b}) &= \mathtt{ba}^{k-i-2} \text{ for all } 4 \leq i \leq k-2,\\
\beta^\star(\mathtt{a}^3\mathtt{b}) &= \mathtt{b}^5(\mathtt{ab})^{k-6}\mathtt{a},\\
\beta^\star(\mathtt{a}^2\mathtt{b}) &= \mathtt{aaba}^{2k-8},\\
\beta^\star(\mathtt{ab}) &= \mathtt{b}^{k-2}\$\mathtt{aba}^{2k-6},\\
\beta^\star(\mathtt{ba}) &= \mathtt{ba}^{k-5}\mathtt{bbbab}^{k-5}\mathtt{ab}^{k-2}\mathtt{a},\\
\beta^\star(\mathtt{b}^j\mathtt{a}) &= \mathtt{bab}^{2k-2j-2}\mathtt{a} \text{ for all } 2 \leq j \leq k-1 \text{ and}\\
\beta^\star(\mathtt{b}^k\mathtt{a}) &= \mathtt{a}.
\end{aligned}$$

Hence, $\mathtt{bwt}(w_k\mathtt{a}\$) = \beta^\star(\$) \cdot (\prod_{i=2}^{k-1} \beta^\star(\mathtt{a}^{k-i}\mathtt{b})) \cdot \beta^\star(\mathtt{b}\$) \cdot (\prod_{i=1}^{k} \beta^\star(\mathtt{b}^i\mathtt{a}))$. Moreover, it holds that $r(w_k\mathtt{a}\$) = 8k - 16$.

Proof. We obtain $\mathtt{bwt}(w_k\mathtt{a}\$) = \mathtt{abwt}(w_k\$)$ by applying Proposition 5.1.18 to the words $w_k\mathtt{a}\$$ and $w_k\$$, and we already know the structure of $\mathtt{bwt}(w_k\$)$ by Lemma 5.1.19. □

With all these intermediates lemmas, we obtain the following.

Proposition 5.1.23 There exists an infinite family of words such that: (i) $r_\$(w\mathtt{b}) - r_\$(w) = \Theta(\sqrt{n})$; (ii) $r_\$(\widehat{w}) - r_\$(w) = \Theta(\sqrt{n})$; (iii) $r_\$(\widehat{w}\mathtt{a}) - r_\$(w) = \Theta(\sqrt{n})$.

Proof. Such a family is composed of the words $w_k\mathtt{b}$ with $k > 5$. The proof follows from Lemma 5.1.19, Lemma 5.1.20, Lemma 5.1.21, Lemma 5.1.22, and Observation 5.1.2. □

5.1.4 The relationship between r and $r_\$$

Now we address the differences between the measures r and $r_\$$. In fact, not only are the measures r and $r_\$$ not equal over the same input, but they may differ by a $\Theta(\log n)$ multiplicative factor, or by a $\Theta(\sqrt{n})$ additive factor [60]. We show the latter result.

Proposition 5.1.24 There exists an infinite family of words w such that $r_\$(w) - r(w) = \Theta(\sqrt{n})$, where $n = |w|$.

Proof. The family consists of the words w_k for all $k > 5$. From Proposition 5.1.13 and Lemma 5.1.19, it holds $r_\$(w_k) - r(w_k) = 2k - 4$. By Observation 5.1.2, it holds $2k - 4 = \Theta(\sqrt{n})$. □

The results in this section suggest that when using BWT-based compressors (both variants), one has to be specially careful when edits operations are performed on the input string. This means that on top of r and $r_\$$ not being ideal as measures of repetitiveness, BWT-based compressors are not ideal for dynamic settings.

5.2 Sensitivity of BWT to Morphism Application

In the field of Combinatorics on Words, morphisms are a fundamental tool for generating repetitive sequences, with multiple applications. For instance, injective morphisms, known as codes, are widely used in the fields of Information Theory, Data Compression, and Cryptography [13]. The relationship between morphisms and the measure r has been studied in the context of a subclass of infinite words generated by morphisms, i.e., the purely morphic words [20, 47]. In this section, we focus on the impact of morphism application on the number of BWT equal-letter runs of finite words.

In Subsection 5.2.1, we explain some specific concepts and present some already known results on *Sturmian morphisms*, which are needed in the remaining of the section.

In Subsection 5.2.2, we prove that a binary morphism is cyclic (i.e., the images of both letters are powers of the same word) if and only if the image of every word under this morphism has the same number of BWT equal-letter runs, regardless of the input word. We also prove other results relating morphisms and words sharing the same Parikh vector (i.e., having the same number of occurrences of each letter), which can be of independent interest.

Then, in Subsection 5.2.3 we find a novel characterization of Sturmian morphisms [14, 97] in terms of BWT equal-letter runs: they are exactly the binary morphisms that preserve the number of BWT equal-letter runs of every binary word containing both letters of the alphabet. This characterization is interesting from a combinatorial point of view, because Sturmian morphisms are a widely studied subject [14, 97]. It also builds another bridge between Combinatorics on Words and Data Compression.

Further, in Subsection 5.2.4 we show a wide class of morphisms, which we call Thue–Morse-like morphisms, that increase the number of BWT equal-letter runs by 2 on every binary word containing both letters of the alphabet. Moreover, for each even number $2k$, we can find a wide class of binary morphisms, obtained by composing Sturmian and Thue–Morse-like morphisms, that increase the BWT equal-letter runs of every binary words by exactly $2k$. Note that this is exhaustive for the binary alphabet. In fact, unless considering powers of a single letter, every binary word has an even number of BWT equal-letter runs. In addition, we can use the aforementioned morphisms to construct arbitrarily large families of binary words having all the same number of BWT equal-letter runs, for every fixed (even) number, and converging to an infinite aperiodic word.

At the other end of the spectrum, in Subsection 5.2.5 we show that there are binary morphisms (in particular, the so-called period-doubling morphism) that can highly increase the number of BWT equal-letter runs of binary words. We show that the increase in the number of BWT equal-letter runs can be $\Omega(\sqrt{n})$, where n is the length of the original word. In Section 5.2.6, we show that this degree of increase cannot occur in other relevant reachable repetitiveness measures, like z or g.

5.2.1 Preliminaries

A morphism $\varphi : \Sigma^* \to \Gamma^*$ is *cyclic* if there exists a word $z \in \Gamma^*$ such that $\varphi(a) \in z^*$, for each $a \in \Sigma$. Otherwise, it is called *acyclic*. Note that the fixed-point of a cyclic morphism is periodic. In the case of a binary morphism, it is known that φ is cyclic if and only if $\varphi(\mathtt{ab}) = \varphi(\mathtt{ba})$.

A *Sturmian morphism* is a morphism that maps infinite Sturmian words into infinite Sturmian words. Some combinatorial characterizations of Sturmian morphisms have been proved [14]. In particular, a binary morphism φ is Sturmian if and only if it is acyclic and *balanced* (i.e., it maps balanced words to balanced words). Berstel and Séébold [14] also proved the following characterization:

Theorem 5.2.1 ([14]) An acyclic morphism φ is Sturmian if and only if it is locally Sturmian, that is, there exists a Sturmian word s such that $\varphi(s)$ is Sturmian.

Let us denote the following morphisms:

$$E : \begin{cases} \mathtt{a} \mapsto \mathtt{b} \\ \mathtt{b} \mapsto \mathtt{a} \end{cases} \qquad \Phi : \begin{cases} \mathtt{a} \mapsto \mathtt{ab} \\ \mathtt{b} \mapsto \mathtt{a} \end{cases} \qquad \tilde{\Phi} : \begin{cases} \mathtt{a} \mapsto \mathtt{ba} \\ \mathtt{b} \mapsto \mathtt{a} \end{cases}$$

The morphism Φ is called the *Fibonacci morphism*, since its fixed point is the Fibonacci word $\mathtt{abaababaabaababaab}\cdots$. The monoid $\{E, \Phi, \tilde{\Phi}\}^*$ generated by E, Φ, and $\tilde{\Phi}$, by using the composition operator $\circ$, is known as the *Sturm monoid*. The following theorem [97], shows the combinatorial structure of Sturmian morphisms.

Theorem 5.2.2 A morphism is Sturmian if and only if it belongs to $\{E, \Phi, \tilde{\Phi}\}^*$.

The Burrows–Wheeler transform is strictly related to the notions of balance, Sturmian word and morphism, as shown in the following proposition.

Proposition 5.2.3 Let w be a word such that $\mathsf{alph}(w) = \{\mathtt{a}, \mathtt{b}\}$. Then the following are equivalent:

1. w is circularly balanced;
2. $w \in \mathcal{R}(s^\ell)$, for some standard Sturmian word s and for some $\ell > 0$;
3. $r(w) = 2$;
4. $w = (\varphi(\mathtt{a}))^\ell$ for a Sturmian morphism φ and for some $\ell > 0$.

Proof. The equivalence of 1, 2 and 3 is in [95, 116]. The equivalence with 4 is in [33] (see also Proposition 10 in [113]). □

5.2.2 Morphisms and sorted rotations of words

We introduce some definitions regarding the rotations of morphic images of words.

Definition 5.2.4 Let $\varphi : \Sigma^* \mapsto \Gamma^*$ be a morphism. Then, we define the *multisets*

$$\mathcal{I}_\varphi(w) = \{\varphi(w') \mid w' \in \mathcal{R}(w)\}$$
$$\mathcal{S}_\varphi(w) = \{v\varphi(w')u \mid u, v \in \Gamma^+, uv = \varphi(a) \text{ for some } a \in \Sigma, \text{ and } aw' \in \mathcal{R}(w)\}.$$

The multiset $\mathcal{I}_\varphi(w)$ corresponds to the rotations of $\varphi(w)$ obtained by applying φ to the rotations of w. The multiset $\mathcal{S}_\varphi(w)$ corresponds to all the remaining rotations of $\varphi(w)$. We refer to the multiset $\mathcal{I}_\varphi(w)$ as the *I-rotations* of $\varphi(w)$, and to the multiset $\mathcal{S}_\varphi(w)$ as the *S-rotations* of $\varphi(w)$. These two multisets could have elements that end up being equal, as we show in the following example.

Example 5.2.5 Let $\varphi \equiv (\mathtt{a}, \mathtt{bab})$, which is an acyclic binary morphism. Then, $\mathtt{ab}$ is primitive but $\varphi(\mathtt{ab}) = \mathtt{abab}$ is not. Moreover, $\mathcal{I}_\varphi(w) = \{\mathtt{abab}, \mathtt{baba}\} = \mathcal{S}_\varphi(w)$.

We now prove some combinatorial properties of words having the same Parikh vector. Note that two rotations of the same word have the same Parikh vector. By using such properties, we prove that, in the case of the binary alphabet, the lexicographic order among the rotations of a given word is either preserved or reversed, after a morphism is applied. This is a key point to show that the number of BWT-runs cannot decrease after the application of a binary morphism. This is no longer true for larger alphabets.

The following lemma shows that distinct words having the same Parikh vector must have Hamming distance of at least 2.

Lemma 5.2.6 Let $w_1, w_2 \in \Sigma^*$ be such that $w_1 \neq w_2$ and $\mathtt{parikh}(w_1) = \mathtt{parikh}(w_2)$. Then, $d_H(w_1, w_2) \geq 2$.

Proof. By definition of d_H, we have that $d_H(w_1, w_2) = 0$ if and only if $w_1 = w_2$. So, let us suppose by contradiction that $d_H(w_1, w_2) = 1$. Then, there exist two finite words $u, v \in \Sigma^*$

and two distinct indices $i, j \in [1..\sigma]$ with $i < j$, such that $w_1 = ua_iv$ and $w_2 = ua_jv$. It follows that the Parikh vectors of w_1 and w_2 are respectively

$$\texttt{parikh}(w_1) = (|u|_{a_1} + |v|_{a_1}, \ldots, |u|_{a_i} + |v|_{a_i} + 1, \ldots, |u|_{a_j} + |v|_{a_j}, \ldots, |u|_{a_\sigma} + |u|_{a_\sigma})$$

and

$$\texttt{parikh}(w_2) = (|u|_{a_1} + |v|_{a_1}, \ldots, |u|_{a_i} + |v|_{a_i}, \ldots, |u|_{a_j} + |v|_{a_j} + 1, \ldots, |u|_{a_\sigma} + |u|_{a_\sigma}).$$

Thus, we obtain that the $\texttt{parikh}(w_1) \neq \texttt{parikh}(w_2)$, a contradiction. □

Since all the words in the same conjugacy class share the same Parikh vector, we can derive the following

Corollary 5.2.7 Let $w \in \Sigma^*$ be a word. Then, for every word $w' \in \mathcal{R}(w)$ such that $w' \neq w$, one has $d_H(w, w') \geq 2$.

Here, we introduce and study new properties of some classes of morphisms, which are related to the number of BWT-runs.

Definition 5.2.8 A morphism φ is *abelian order-preserving* if for every pair of distinct words x and y having the same Parikh vector, it holds that $x < y \iff \varphi(x) < \varphi(y)$.

Definition 5.2.9 A morphism φ is *abelian order-reversing* if for every pair of distinct words x and y having the same Parikh vector, it holds that $x < y \iff \varphi(x) > \varphi(y)$.

In general, a morphism can be neither abelian order-preserving nor abelian order-reversing:

Example 5.2.10 A cyclic morphism is trivially not abelian order-preserving nor abelian order-reversing. The acyclic morphism $\varphi \equiv (\mathtt{b}, \mathtt{a}, \mathtt{c})$ is also neither of them. This can be verified on the rotations of the words $\mathtt{abc}$ and $\varphi(\mathtt{abc}) = \mathtt{bac}$.

However, all acyclic morphisms with a binary domain are either abelian order-preserving or abelian order-reversing, as we show in the following lemma.

Lemma 5.2.11 Let $\varphi : \{\mathtt{a}, \mathtt{b}\}^* \mapsto \Sigma^*$ be an acyclic morphism. Then, φ is either abelian order-preserving or abelian order-reversing.

Proof. Let $\varphi \equiv (\alpha, \beta)$ be an acyclic morphism (i.e., $\alpha\beta \neq \beta\alpha$). For the proof, we assume that $|\alpha| \leq |\beta|$, and the other case is treated symmetrically. Factorize φ as $(\alpha, \beta) = (\alpha, \alpha^k v)$, where $k \geq 0$ is as big as possible. This factorization is unique, and α is not a prefix of v, otherwise, k is not as big as possible. Also, $v \neq \varepsilon$ and $v \neq \alpha$ because the morphism φ is acyclic. Let $x = u\mathtt{a}z_1$ and $y = u\mathtt{b}z_2$ be two distinct binary words with the same Parikh vector. Note that a $\mathtt{b}$ has to appear in z_1, since otherwise x has fewer $\mathtt{b}$'s than y. Let $z_1 = \mathtt{a}^t bz_1'$ for some $t \geq 0$ and $z_1' \in \{\mathtt{a}, \mathtt{b}\}^*$. We can write $x = u\mathtt{a}\mathtt{a}^t bz_1'$. Then, $\varphi(x) = \varphi(u)\alpha^k\alpha\alpha^t v\varphi(z_1')$ and $\varphi(y) = \varphi(u)\alpha^k v\varphi(z_2)$. We proceed by case analysis.

If v is not a prefix of α, then the order between $\varphi(x)$ and $\varphi(y)$ depends only on the order between α and v. The reason is that $\varphi(x)$ and $\varphi(y)$ share a common prefix $\varphi(u)\alpha^k$, followed by α and v respectively, which differ at some position from left to right. Hence, if $\alpha < v$, we obtain $x < y \iff \varphi(x) < \varphi(y)$; if $v < \alpha$, then we obtain $x < y \iff \varphi(x) > \varphi(y)$.

If v is a proper prefix of α and $k > 0$, rewrite $\varphi(y) = \varphi(u)\alpha^k v\alpha z_2'$. We can do this because y has to have at least one letter after $u\mathtt{b}$ and both images α and β start with α (in the case of β because $k > 0$). We note that the common prefix $\varphi(u)\alpha^k$ is followed by αv in $\varphi(x)$ (αv is a prefix of $\alpha\alpha$), and by $v\alpha$ in the case of $\varphi(y)$. The order between $\varphi(x)$ and $\varphi(y)$ is then completely determined by the order between αv and $v\alpha$. This happens because αv and $v\alpha$ are words of the same length which must be distinct, as implied by the inequality $\alpha\beta = \alpha\alpha^k v \neq \beta\alpha = \alpha^k v\alpha$. Hence, if $\alpha v < v\alpha$, we obtain $x < y \iff \varphi(x) < \varphi(y)$; if $v\alpha < \alpha v$, then we obtain $x < y \iff \varphi(x) > \varphi(y)$.

No other case is possible. By construction, α is not a prefix of v. Also, $\alpha \neq v$, so if v is a prefix of α, it has to be a proper prefix. If this is the case, as $|\alpha| \leq |\alpha^k v|$ and $|v| < |\alpha|$, k has to be at least 1. □

Using Lemma 5.2.11 we can easily derive the following corollary.

Corollary 5.2.12 Let w be a binary word and let φ be an acyclic morphism. Then, for all pairs of rotations u, v of w, either $u < v \iff \varphi(u) < \varphi(v)$ (when φ is abelian order-preserving), or $u < v \iff \varphi(u) > \varphi(v)$ (when φ is abelian order-reversing).

We introduce new measures to study how the action of a morphism affects the BWT-runs.

Definition 5.2.13 Let φ be a morphism and w a word. We define

$$\Delta_\varphi^+(w) = r(\varphi(w)) - r(w)$$

and

$$\Delta_\varphi^\times(w) = \frac{r(\varphi(w))}{r(w)}.$$

Acyclic binary morphisms cannot decrease the number of BWT-runs of any word.

Theorem 5.2.14 Let $\varphi : \{\mathtt{a}, \mathtt{b}\}^* \mapsto \Sigma^*$ be an acyclic morphism. Then $\Delta_\varphi^+(w) \geq 0$ for every $w \in \{\mathtt{a}, \mathtt{b}\}^*$.

Proof. Let $\varphi \equiv (\alpha, \beta)$. Since $r(w) = r(w^m)$ for every $w \in \Sigma^*$ and $m > 1$, let us assume that w is primitive. For the proof, we assume that $|\alpha| \geq |\beta|$, and the other case is treated symmetrically. First, let us consider the case where β is not a suffix of α. Let moreover $x \in \Sigma^*$ be the longest common suffix between α and β. It follows that there exist $\alpha', \beta' \in \Sigma^+$ such that $\alpha = \alpha' x$ and $\beta = \beta' x$, and that the last symbol of α' is different from the last of β' (otherwise x would be longer). Let $\mathcal{R}_x(\varphi(w))$ denote the multiset of rotations of $\varphi(w)$ with x as a prefix. Note that if $x = \varepsilon$, then $\mathcal{R}_x(\varphi(w)) = \mathcal{I}_\varphi(w)$. Since x appears in both α and β, it follows that $|\mathcal{R}_x(\varphi(w))| \geq |w|$. Specifically, for each $i \in [1\,..\,|w|]$, there exists $t_i \in \mathcal{R}_x(\varphi(w))$ such that $t_i = x\varphi(w[i+1\,..\,|w|] \cdot w[1\,..\,i-1])v$, where v is either α' or β', depending on whether $w[i]$ is $\mathtt{a}$ or $\mathtt{b}$ respectively. The lexicographical order of these $|w|$ rotations of $\varphi(w)$ with the same prefix correspond to the lexicographical order of the rotations in $\mathcal{I}_\varphi(w)$, since by Corollary 5.2.7 the words $\bigcup_{i=1}^{|w|}\{\varphi(w[i+1,|w|]\cdot w[1\,..\,i-1])\}$ must differ in at least one position. By Corollary 5.2.12 this is either in the same or in the reverse order with respect to the sorting of the rotations of w. Thus, there exists an injective coding $\lambda : \{\mathtt{a},\mathtt{b}\}^* \mapsto \Sigma'^* \subseteq \Sigma$ such that either $\lambda(\mathtt{bwt}(w))$ or $\lambda(\mathtt{bwt}(w)^R)$ is a subsequence of $\mathtt{bwt}(\varphi(w))$, and therefore $r(\varphi(w)) \geq r(w)$.

Let us now consider the case where β is suffix of α. Then, there exists a primitive word $u \in \Sigma^+$ and two integers $p \geq q \geq 1$ such that $\beta = u^q$, and $\alpha = \alpha' u^p$, with $\alpha' \in \Sigma^+$ that does not have u as suffix. Note that $\alpha' \neq \varepsilon$, otherwise we would have $\alpha\beta = u^p u^q = u^q u^p = \beta\alpha$, i.e. φ would not be acyclic. Let x be the longest common suffix between α' and u. If $x \neq \alpha'$, from analogous arguments to the case where β is not a suffix of α, we have at least $r(w)$ equal-letter runs in $\mathcal{R}_{xu^p}(\varphi(w))$. Otherwise, if $x = \alpha'$, let us consider the word $y \in \Sigma^+$ such that $u = yx$. We can then consider the longest common suffix x' between xy and yx, which must be a proper suffix (otherwise u would not be primitive), and apply the same reasoning over the set $\mathcal{R}_{x'xu^p}(\varphi(w))$ and the thesis follows. □

The following example shows that Theorem 5.2.14 does not hold in the case of larger alphabets.

Example 5.2.15 Consider the acyclic morphism $\varphi \equiv (\mathtt{b},\mathtt{a},\mathtt{c})$. Then, $\mathtt{bwt}(\mathtt{bcba}) = \mathtt{bcab}$ and $\mathtt{bwt}(\varphi(\mathtt{bcba})) = \mathtt{bwt}(\mathtt{acab}) = \mathtt{cbaa}$.

An immediate consequence of Theorem 5.2.14 is the following.

Corollary 5.2.16 Let $\varphi : \{\mathtt{a},\mathtt{b}\}^* \mapsto \Sigma^*$ be an acyclic morphism. Then, $\Delta^{\times}_{\varphi}(w) \geq 1$, for every $w \in \{\mathtt{a},\mathtt{b}\}^*$.

The following theorem provides a characterization of cyclic morphisms in terms of the number of BWT-runs.

Theorem 5.2.17 A morphism $\varphi : \{\mathtt{a},\mathtt{b}\}^* \mapsto \Sigma^*$ is cyclic if and only if there exists $k > 0$ such that $r(\varphi(w)) = k$ for all $w \in \{\mathtt{a},\mathtt{b}\}^*$.

Proof. If $\varphi \equiv (\alpha,\beta)$ is cyclic then there exists a primitive word $u \in \Sigma^*$ such that $\alpha = u^p$ and $\beta = u^q$, for some $p, q \geq 0$. Therefore, for each word $w \in \{a,b\}^*$, we have $r(\varphi(w)) = r(u^{p\cdot|w|_{\mathtt{a}}+q\cdot|w|_{\mathtt{b}}}) = r(u)$. The other implication is a consequence of Theorem 5.2.14. In fact, by contraposition for each $k > 0$ we can find a word w such that $r(w) > k$ (for instance, the i-th Thue–Morse finite word such that $i > \frac{k}{2}$ [20]), which leads to $r(\varphi(w)) \geq r(w) > k$ as well. □

5.2.3 Binary morphisms preserving r

This section is devoted to characterizing binary morphisms such that the number of BWT equal-letter runs is preserved after the action of the morphism on any binary word. First, we show with an example that this property is not trivial.

Example 5.2.18 Let $\theta \equiv (\mathtt{ab}, \mathtt{aa})$ be the period-doubling morphism. It can be verified that $\Delta^+_\theta(\mathtt{ab}) = 0$, $\Delta^+_\theta(\mathtt{aab}) = 2$, and $\Delta^+_\theta(\mathtt{aaabbaabab}) = 4$.

Next, we show that every Sturmian morphism fixes the number of BWT-runs. From the definition of E, Φ, and $\tilde{\Phi}$, and by Lemma 5.2.11, we derive the following.

Lemma 5.2.19 Let $w \in \{\mathtt{a}, \mathtt{b}\}^*$ be a binary word. Then, for all pairs of rotations u and v of w, and for each $\chi \in \{E, \Phi, \tilde{\Phi}\}$, it holds that $u < v$ if and only if $\chi(u) > \chi(v)$.

We prove that the number of BWT-runs is preserved by the morphisms that are the generators of the Sturmian morphisms. Note that from the following lemma a method can be derived to construct $\mathtt{bwt}(\varphi(w))$ starting from $\mathtt{bwt}(w)$, for every Sturmian morphism φ and every binary word w.

Lemma 5.2.20 Let $w \in \{\mathtt{a}, \mathtt{b}\}^*$ be a binary word with $|\mathsf{alph}(w)| = 2$. Then, for all $\chi \in \{E, \Phi, \tilde{\Phi}\}$, one has $r(w) = r(\chi(w))$. More in detail, one has $\mathtt{bwt}(E(w)) = \overline{\mathtt{bwt}(w)^R}$ and $\mathtt{bwt}(\Phi(w)) = \mathtt{bwt}(\tilde{\Phi}(w)) = \overline{\mathtt{bwt}(w)^R} \cdot \mathtt{a}^{|w|_\mathtt{a}}$.

Proof. Since for each word w and each integer $k > 0$ we have $r(w) = r(w^k)$, let us assume that w is a primitive word. From Lemma 5.2.19, the case $\chi = E$ is trivial: in fact, from it follows that $\mathtt{bwt}(E(w)) = \overline{\mathtt{bwt}(w)^R}$, and therefore $r(w) = r(E(w))$.

For the case $\chi = \Phi$ one can observe that every $\mathtt{b}$ that occurs in $\Phi(w)$ is obtained from $\Phi(\mathtt{a})$, and therefore it is always preceded by an $\mathtt{a}$. Thus, the rotations of $\Phi(w)$ left to cover are all those starting with an $\mathtt{a}$, which therefore must also start with either $\Phi(\mathtt{a})$ or $\Phi(\mathtt{b})$. By Lemma 5.2.19, and by observing that $\Phi(\mathtt{a})$ ends with a $\mathtt{b}$ and $\Phi(\mathtt{b})$ ends with an $\mathtt{a}$, we have that $\mathtt{bwt}(\Phi(w)) = \overline{\mathtt{bwt}(w)^R} \cdot \mathtt{a}^{|w|_\mathtt{a}}$. Thus, we need to check if the run of $\mathtt{a}$'s at the end merges with the last symbol of $\overline{\mathtt{bwt}(w)^R}$. This is equivalent to checking that the first symbol of $\mathtt{bwt}(w)$ is a $\mathtt{b}$, and by contradiction if the first rotation in lexicographical order is $u\mathtt{a}$ for some $u \in \{\mathtt{a}, \mathtt{b}\}^{n-1}$, then $\mathtt{a}u$ is a conjugate of w and $\mathtt{a}u < u\mathtt{a}$ for each binary word w, a contradiction.

For the case $\chi = \tilde{\Phi}$, one can see for any binary word $w = w_1 w_2 \cdots w_n$ we have that $\Phi(w) = \Phi(w_1 w_2 \cdots w_n) = \mathtt{a}v_1\mathtt{a}v_2 \cdots \mathtt{a}v_n$, where for each $i \in [1\,..\,n]$ we have $v_i = \mathtt{b}$ if $w_i = \mathtt{a}$, or $v_i = \varepsilon$ if $w_i = \mathtt{b}$. On the other hand, for the same word w we have $\tilde{\Phi}(w) = \tilde{\Phi}(w_1 w_2 \cdots w_n) = v_1\mathtt{a}v_2 \cdots \mathtt{a}v_n\mathtt{a}$, where analogously to the previous case $v_i = \mathtt{b}$ if $w_i = \mathtt{a}$, or $v_i = \varepsilon$ if $w_i = \mathtt{b}$. One can notice that $\Phi(w)$ and $\tilde{\Phi}(w)$ are conjugate, and the thesis follows. □

A graphical interpretation of Lemma 5.2.20 is shown in Figure 5.3.

The following theorem shows a new characterization of Sturmian morphisms.

Theorem 5.2.21 Let φ be a binary morphism. Then, the following are equivalent:

1. $\Delta^+_\varphi(w) = 0$ for every word w with $|\mathsf{alph}(w)| = 2$;
2. φ is a Sturmian morphism.

$M(w)$

aabba	b
abaab	b
abbab	a
baabb	a
babaa	b
bbaba	a

$M(\Phi(w))$

a.a.ab.a.ab.a	**b.**
a.ab.a.ab.ab.	**a.**
a.ab.ab.a.a.a	**b.**
ab.a.a.ab.a.a	**b.**
ab.a.ab.ab.a.	**a.**
ab.ab.a.a.ab.	**a.**
b.a.a.ab.a.ab.	a
b.a.ab.ab.a.a.	a
b.ab.a.a.ab.a.	a

$M(\tilde{\Phi}(w))$

a.	a.a.ba.a.ba.	b
a.	**a.ba.a.ba.b**	**a.**
a.	a.ba.ba.a.a.	b
a.	ba.a.a.ba.a.	b
a.	**ba.a.ba.ba.**	**a.**
a.	**ba.ba.a.a.b**	**a.**
b	**a.a.a.ba.a.b**	**a.**
b	**a.a.ba.ba.a.**	**a.**
b	**a.ba.a.a.ba.**	**a.**

Figure 5.3: From left to right, the BWT matrix for the words w = abbaba, $\Phi(w)$, and $\tilde{\Phi}(w)$ respectively. For $M(\Phi(w))$ and $M(\tilde{\Phi}(w))$, we separate with dots the images of symbols from w. The rotations in bold of $M(\Phi(w))$ and $M(\tilde{\Phi}(w))$ correspond to the words in $\mathcal{I}_\Phi(w)$ and $\mathcal{I}_{\tilde{\Phi}}(w)$ respectively. The block of rotations in gray at the end of both $M(\Phi(w))$ and $M(\tilde{\Phi}(w))$ are in correspondence of the equal-letter run of a's of length $|w|_{\mathtt{a}}$, which occurs for every $w \in \{\mathtt{a}, \mathtt{b}\}^*$. One can see that $\mathtt{bwt}(\Phi(w)) = \mathtt{bwt}(\tilde{\Phi}(w)) = \overline{\mathtt{bwt}(w)^R} \cdot \mathtt{a}^{|w|_{\mathtt{a}}}$.

Proof. By Theorem 5.2.2 and Lemma 5.2.20, all Sturmian morphisms preserve the number of BWT-runs. Conversely, suppose that φ preserves the number of BWT-runs. By Theorem 5.2.17, such a morphism must be acyclic. Let $s = \lim s_i$ be a characteristic Sturmian word. For every i, the word $\varphi(s_i)$ has 2 runs in its BWT, hence it is circularly balanced (Proposition 5.2.3). Let us consider the word $\varphi(s) = \lim \varphi(s_i)$. It is balanced and aperiodic, since it is obtained by applying an acyclic morphism to a Sturmian word [25]. Then, $\varphi(s)$ is Sturmian by using Theorem 2.4.2, whence φ is a Sturmian morphism by applying Theorem 5.2.1. □

5.2.4 Binary morphisms increasing r by a constant

The next step after characterizing Sturmian morphisms as those fixing BWT equal-letter runs on binary words, is to find other binary morphisms that increase the number of BWT-runs always by the same fixed constant. Remind that if such a constant exists, it has to be an even integer because the BWT of any binary word starts with b and ends with a.

We show that for every $k > 0$, we can find a morphism increasing the BWT-runs of any binary word by exactly $2k$. We do so by showing a family of binary morphisms that increase the BWT-runs always by 2, which then we can compose as we want. This family is formed by binary morphisms that are similar to the famous Thue–Morse morphism $\tau \equiv (\mathtt{ab}, \mathtt{ba})$. The structure of the BWT of Thue–Morse words has been studied before and it is well understood [20, 36]. We generalize such results by showing how to derive $\mathtt{bwt}(\varphi(w))$ from $\mathtt{bwt}(w)$ for every Thue–Morse-like morphism φ and every binary word w.

Definition 5.2.22 A binary morphism is *Thue–Morse-like* if it has the form $\tau_{p,q} \equiv (\mathtt{ab}^p, \mathtt{ba}^q)$ for some $p, q > 0$.

We prove the following proposition, which is crucial to obtain the main result of this section. Figure 5.4 highlights the key aspects of the proof.

Proposition 5.2.23 For every binary word w such that $\mathsf{alph}(w) = \{\mathtt{a}, \mathtt{b}\}$, the I-rotations of $\tau_{p,q}(w)$ are contiguous in the BWT matrix of $\tau_{p,q}(w)$, and their last letters spell $\overline{\mathtt{bwt}(w)}$.

Proof. Let w be a binary word of length n such that $\mathsf{alph}(w) = \{\mathtt{a}, \mathtt{b}\}$. Observe that $\tau_{p,q} \equiv (\mathtt{ab}^p, \mathtt{ba}^q)$ is abelian order-preserving, so the I-rotations of $\tau_{p,q}(w)$ maintain their relative order. Because $\tau_{p,q}(\mathtt{a})$ ends with $\mathtt{b}$ and $\tau_{p,q}(\mathtt{b})$ ends with $\mathtt{a}$, if we consider only the I-rotations of $\tau_{p,q}(w)$ and take the last letter of each, we obtain $\overline{\mathtt{bwt}(w)}$, which starts with $\mathtt{a}$ and ends with $\mathtt{b}$. It remains to show that all the I-rotations of $\tau_{p,q}(w)$ are contiguous in its BWT matrix.

If $p > 1$, each S-rotation starting with $\mathtt{a}$, has to start either with $\mathtt{a}^i\mathtt{b}$ for some $2 \leq i \leq q+1$, or with $\mathtt{aba}^q$, and both of these prefixes are smaller than $\mathtt{ab}^p$. If $p = 1$, an S-rotation starting with $\mathtt{a}$ is smaller than the word $\mathtt{a}(\mathtt{ba}^q)^{n-1}\mathtt{ba}^{q-1}$, which is smaller than a rotation having $(\mathtt{ab})^i\mathtt{ba}$ as a prefix for some $0 < i < n$. The I-rotations that start with $\mathtt{a}$ have prefixes of such type. In both cases, we obtain that the S-rotations starting with $\mathtt{a}$ are smaller than the I-rotations starting with $\mathtt{a}$. A symmetric argument shows that S-rotations starting with $\mathtt{b}$ are greater than the I-rotations starting with $\mathtt{b}$. Thus, the I-rotations are contiguous and the thesis holds. □

Now we are ready to show that Thue–Morse-like morphisms increase the number of BWT-runs of binary words always by 2.

Lemma 5.2.24 For every binary word w such that $\mathsf{alph}(w) = \{\mathtt{a}, \mathtt{b}\}$, it holds that

$$\mathtt{bwt}(\tau_{p,q}(w)) = \mathtt{b}^{|w|_\mathtt{b}}\mathtt{a}^{(q-1)|w|_\mathtt{b}} \cdot \overline{\mathtt{bwt}(w)} \cdot \mathtt{b}^{(p-1)|w|_\mathtt{a}}\mathtt{a}^{|w|_\mathtt{a}},$$

and that $r(\tau_{p,q}(w)) = r(w) + 2$.

Proof. We show that the block of $\mathtt{bwt}(\tau_{p,q}(w))$ that corresponds to the S-rotations starting with the letter $\mathtt{a}$ is equal to $\mathtt{b}^{|w|_\mathtt{b}}\mathtt{a}^{(q-1)|w|_\mathtt{b}}$. If $q = 1$, all the S-rotations starting with $\mathtt{a}$ end with the letter $\mathtt{b}$. If $q > 1$, the only S-rotations that start with $\mathtt{a}$ and end with $\mathtt{b}$ have as a prefix either $\mathtt{a}^{q+1}\mathtt{b}$ or $\mathtt{a}^q\mathtt{ba}^q$. The smallest S-rotation starting with $\mathtt{a}$ and ending with $\mathtt{a}$ starts with $\mathtt{a}^q\mathtt{b}^p\mathtt{ab}$ or $\mathtt{a}^q\mathtt{b}^p\mathtt{ba}$. Hence, S-rotations starting with $\mathtt{a}$ and ending with $\mathtt{b}$ appear before those ending with $\mathtt{a}$.

It follows that the block of $\mathtt{bwt}(\tau_{p,q}(w))$ defined by the S-rotations starting with $\mathtt{a}$ spells $\mathtt{b}^{|w|_\mathtt{b}}\mathtt{a}^{(q-1)|w|_\mathtt{b}}$, because of their order, and because each of these rotations is in correspondence with some specific $\mathtt{a}$ inside $\tau_{p,q}(\mathtt{b})$ for some specific $\mathtt{b}$ of w. Only one of these $\mathtt{a}$'s per image produces a rotation ending with $\mathtt{b}$, and the other $q-1$ $\mathtt{a}$'s yield rotations ending with $\mathtt{a}$.

Showing that the block of $\mathtt{bwt}(\tau_{p,q}(w))$ corresponding to the S-rotations starting with the letter $\mathtt{b}$ equals $\mathtt{b}^{(p-1)|w|_\mathtt{a}}\mathtt{a}^{|w|_\mathtt{a}}$ is handled symmetrically.

By using Proposition 5.2.23, we obtain

$$\mathtt{bwt}(\tau_{p,q}(w)) = \mathtt{b}^{|w|_\mathtt{b}}\mathtt{a}^{(q-1)|w|_\mathtt{b}} \cdot \overline{\mathtt{bwt}(w)} \cdot b^{(p-1)|w|_\mathtt{a}}\mathtt{a}^{|w|_\mathtt{a}}.$$

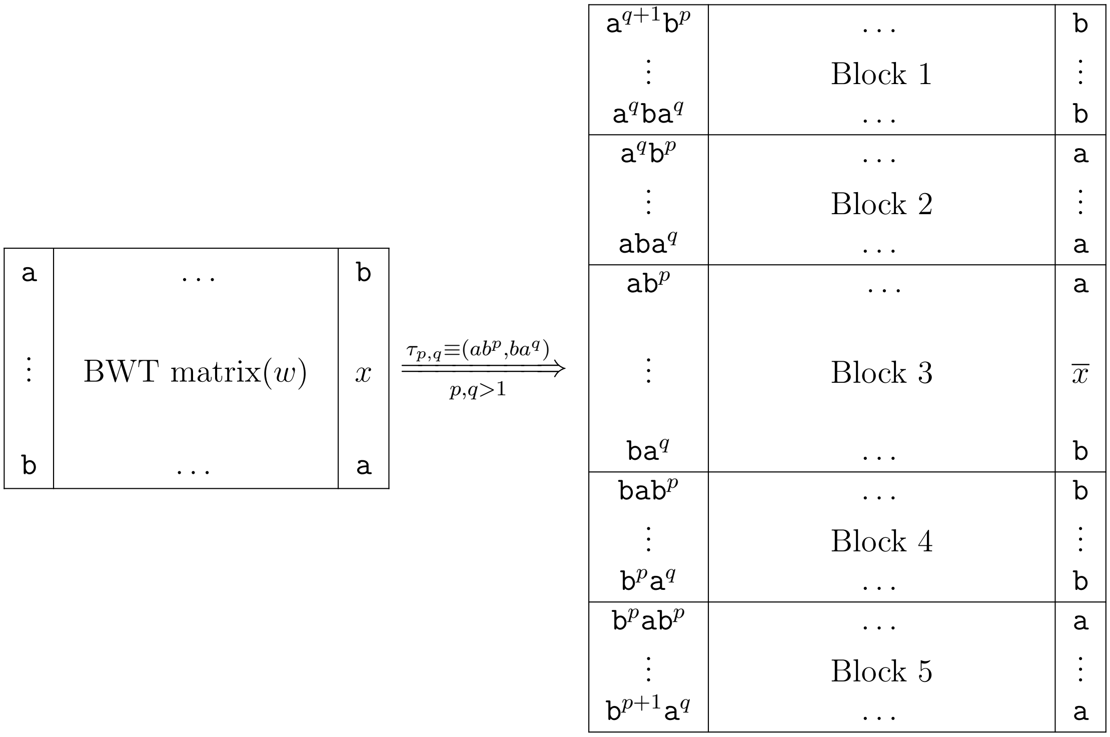

a
...
b
⋮
BWT matrix(w)
x
b
...
a
$\tau_{p,q}\equiv(ab^p,ba^q)$
$p,q>1$
$\mathtt{a}^{q+1}\mathtt{b}^p$
Block 1
$\mathtt{a}^q\mathtt{b}\mathtt{a}^q$
$\mathtt{a}^q\mathtt{b}^p$
Block 2
$\mathtt{aba}^q$
$\mathtt{ab}^p$
Block 3
$\overline{x}$
$\mathtt{ba}^q$
$\mathtt{bab}^p$
Block 4
$\mathtt{b}^p\mathtt{a}^q$
$\mathtt{b}^p\mathtt{ab}^p$
Block 5
$\mathtt{b}^{p+1}\mathtt{a}^q$

Figure 5.4: Scheme showing the action of a Thue–Morse-like morphism $\tau_{p,q} \equiv (\mathtt{ab}^p, \mathtt{ba}^q)$ with $p, q > 1$ on a binary word w with $\mathsf{alph}(w) = \{\mathtt{a}, \mathtt{b}\}$. At the left is the BWT matrix of w. At the right is the BWT matrix of $\tau_{p,q}(w)$. The cases where $p = 1$ or $q = 1$ are similar with Block 2 or Block 4 omitted.

As $\overline{\mathtt{bwt}(w)}$ starts with a and ends with b, we have that $r(\tau_{p,q}(w)) = r(w) + 2$, and the thesis holds. □

As a consequence of Theorem 5.2.21 and Lemma 5.2.24, we obtain the following corollary.

Corollary 5.2.25 Given a non-negative even integer $2t$, there exists a binary morphism φ such that $\Delta^+_\varphi(w) = 2t$ and $\Delta^\times_\varphi(w) \leq t + 1$, for every word w with $|\mathsf{alph}(w)| = 2$.

Proof. We can construct the morphism $\varphi \in (\{E, \Phi, \tilde{\Phi}\} \cup \{(\mathtt{ab}^p, \mathtt{ba}^q) \mid p, q > 0\})^*$ such that φ is obtained by composing, in any order, exactly t morphisms taken in the set $\{(\mathtt{ab}^p, \mathtt{ba}^q) \mid p, q > 0\}$ and an arbitrary number of Sturmian morphisms. By Theorem 5.2.21 and Lemma 5.2.24, it holds that $\Delta^+_\varphi(w) = 2t$. The value of the function $\Delta^\times_\varphi(w) = (r(w)+2t)/r(w) = 1+2t/r(w)$ is maximized when $r(w) = 2$. This maximum is $\Delta^\times_\varphi(w) = t + 1$. □

We conclude this section by showing a simple algorithm that allows us to construct an arbitrarily large family of words $w_1, w_2, \ldots$ with exactly $2t$ BWT-runs each. In Algorithm 1, a morphism φ such that $\Delta^+_\varphi(w) = 2(t - 1)$ for every binary word is required. Note that Corollary 5.2.25 assures that such a morphism exists.

Moreover, each word w_i is a prefix of the next word w_{i+1}, so that the infinite word $w = \lim_{i\to\infty} w_i$ is well defined, and it is aperiodic. This is given because it holds for the

Algorithm 1 Algorithm for constructing words with $2t$ BWT-runs

Input : A morphism φ with $\Delta^{+}_{\varphi}(w) = 2(t-1)$. A sequence of positive integers $d_1, \ldots, d_k$.
Output: A sequence of words $w_1, w_2, \ldots, w_k$ where $r(w_i) = 2t$ for any $1 \leq i \leq k$.

```
$w_{-1} \longleftarrow \varphi(\mathtt{b})$
$w_0 \longleftarrow \varphi(\mathtt{a})$
for $i \in [1\,..\,k]$ do
    $w_i \longleftarrow w_{i-1}^{d_i} w_{i-2}$
return $w_1, \ldots, w_k$
```

(implicit) standard Sturmian words s_i for $i \in [1\,..\,k]$ being used, which are circularly balanced (i.e., $r(w) = 2$ on them, as reported in Proposition 5.2.3), and their limit is a characteristic Sturmian word, which is aperiodic.

5.2.5 Morphisms with an unbounded increase on r

There exist morphisms that do not behave as well as Sturmian and Thue–Morse-like morphisms with respect to r. If we consider an alphabet of size greater than 2, we can always find a morphism φ such that the values $\Delta^{+}_{\varphi}(w)$ and $\Delta^{\times}_{\varphi}(w)$ are arbitrarily large.

Lemma 5.2.26 Let $\Sigma = \{c_1, \ldots, c_k, \mathtt{a}, \mathtt{b}\}$ with $k \geq 1$. Let $\Phi \equiv (\mathtt{ab}, \mathtt{a})$ be the Fibonacci morphism. Then, $r(w) = k+3$ if w belongs to $\{\Phi^{2i}(\mathtt{a})c_1c_2\cdots c_k \mid i \geq 1\}$.

Proof. We prove the result by induction on $k \geq 1$. Observe that the words $\Phi^{2i}(\mathtt{a})$ for $i \geq 1$ are Fibonacci words ending with the letter $\mathtt{a}$. It is known that in these words, if we append the letter c_1 smaller than $\mathtt{a}$ at the end, then the number of runs becomes 4 [102, Theorem 11]. For the inductive step, suppose that $r(\Phi^{2i}(\mathtt{a})c_1 \ldots c_{k-1}) = k+2$. When appending c_k at the end, the rotations that do not start with c_k keep their relative order, and the rotation that originally ended with c_{k-1} now ends with c_k. Hence, they define the same number of runs as before. The rotation starting with c_k can be found after the rotation starting with c_{k-1}, which does not end with $\mathtt{b}$, and before the first rotation starting with $\mathtt{a}$, which ends with $\mathtt{b}$. Hence, the number of runs increases by 1. Thus, $r(\Phi^{2i}(\mathtt{a})c_1 \ldots c_k) = k+3$. □

Lemma 5.2.27 Let $\Sigma = \{c_1, \ldots, c_k, \mathtt{a}, \mathtt{b}\}$ with $k \geq 1$. Let $\Phi \equiv (\mathtt{ab}, \mathtt{a})$ be the Fibonacci morphism, and $\varphi \equiv (c_1, c_2, \ldots, c_k, \mathtt{ab}, \mathtt{a})$ be a morphism on the alphabet Σ. Then, $r(w) = \Omega(\log n)$ for every $w \in \{\varphi(\Phi^{2i}(\mathtt{a})c_1c_2\cdots c_k) \mid i \geq 1\}$.

Proof. The morphism φ maps a Fibonacci word ending with $\mathtt{a}$ having $c_1 \ldots c_k$ appended at the end, to the next Fibonacci word, which ends with $\mathtt{b}$, having $c_1 \ldots c_k$ appended at the end. For $k = 1$, it is known that the number of runs in this family is $\Omega(\log n)$ [60]. In a similar way to Lemma 5.2.26, it is possible to prove by induction that appending c_k at the end of $\varphi(\Phi^{2i}(\mathtt{a})c_1c_2c_{k-1})$ adds 2 runs when $k = 2$ and exactly 1 new run when $k > 2$. □

Proposition 5.2.28 For each alphabet Σ with size greater than 2 there exist a morphism φ, satisfying that for every k, there is a word $w \in \Sigma^*$ such that $\Delta^{+}_{\varphi}(w) \geq k$ and $\Delta^{\times}_{\varphi}(w) \geq k$.

Proof. This is immediate from Lemma 5.2.26 together with Lemma 5.2.27. □

Finding examples like the previous ones for binary morphisms is trickier, but at least in the case of Δ^+_φ, it is possible. An example of a binary morphism for which the value $\Delta^+_\varphi(w)$ can be arbitrarily large is the *period-doubling morphism* denoted by θ and defined by the rules $\theta(\mathtt{a}) = \mathtt{ab}$ and $\theta(\mathtt{b}) = \mathtt{aa}$.

Lemma 5.2.29 Let θ be the period-doubling morphism. For any positive integer k there exist a word w such that $\Delta^+_\theta(w) > k$.

Proof. W.l.o.g assume that $k > 2$. For $i \in [2\,..\,k]$ define the words

$$s_i = \mathtt{ab}^i\mathtt{a} \cdot u_i \text{ and } e_i = \mathtt{ab}^i\mathtt{a} \cdot u_i^R, \text{ where } u_i = \mathtt{a}^{2k-i}\mathtt{ba}^{i-2}$$

We say that s_i is a *starting factor*, and e_i is an *ending factor*. Observe that s_i (resp. u_i) is always smaller than e_i (resp. u_i^R). Moreover, it holds that if $i < j$, then $u_i < u_j < u_j^R < u_i^R$. We define the word $z_k = (\Pi_{i=2}^k s_i e_i)\mathtt{a}^k$ and show that $\Delta^+_\theta(z_k) = 2k$. Figure 5.5 shows the structure of both BWTs and highlights the increase in the number of runs.

Consider the rotations of z_k starting with $\mathtt{b}^i\mathtt{a}$ with $1 < i \leq k$. The left shift of the unique rotation starting with the i-th starting factor, and the left shift of the unique rotation starting with the i-th ending factor, are the smallest and greater, respectively. Both of them end with the letter $\mathtt{a}$. The remaining rotations starting with $\mathtt{b}^i\mathtt{a}$ (if any) have to end with $\mathtt{b}$ because in them the prefix $\mathtt{b}^i\mathtt{a}$ corresponds to a suffix of a longer run of $\mathtt{b}$'s followed by an $\mathtt{a}$.

In the case of the rotations of z_k starting with $\mathtt{ba}$, the one starting in the last $\mathtt{b}$ of e_k, has $\mathtt{ba}^k\mathtt{a}^k$ as a prefix, so it is the smallest of them. Also, this rotation is preceded by the factor $\mathtt{ab}^k\mathtt{aa}^{k-2}$, which ends in $\mathtt{a}$. The greatest rotation starting with $\mathtt{ba}$ is the one starting with the $\mathtt{b}$ preceding e_2, which is followed by $\mathtt{abb}$ and preceded also by an $\mathtt{a}$. In the case of the rotations of z_k starting with $\mathtt{a}$, the smallest of them ends with the letter $\mathtt{b}$ as in any binary word. The greatest is the rotation starting with $\mathtt{ab}^k\mathtt{a}u_k^R$ which is preceded by the letter $\mathtt{a}$.

With the general structure of the BWT of z_k in mind, now we analyze the BWT of $\theta(z_k)$. The morphism θ is order-reversing and all the I-rotations of $\theta(z_k)$ start with the letter $\mathtt{a}$. S-rotations of $\theta(z_k)$ starting with the letter $\mathtt{b}$ are always preceded by an $\mathtt{a}$, and it is easy to see that this run of $\mathtt{a}$'s merges with the last $\mathtt{a}$ in the greatest I-rotation. The S-rotations starting with an $\mathtt{a}$ have an even number of $\mathtt{a}$'s before the first $\mathtt{b}$ appears, and also end with the letter $\mathtt{a}$. This implies that they appear grouped after all the I-rotations of the form $(\mathtt{aa})^i\mathtt{ab}$ for some $1 \leq i \leq k$, and before all the I-rotations starting with $(\mathtt{aa})^{i-1}\mathtt{ab}$. As the smallest and greatest rotations of each of these blocks of I-rotations end with $\mathtt{b}$ (because of the action of θ), it follows that the group of S-rotations starting with $(\mathtt{aa})^i\mathtt{b}$ increases the number of runs of the BWT of $\theta(z_k)$ by 2 with respect to the BWT of z_k. This happens for $1 \leq i \leq k$, so the overall increase in r after applying the morphism θ is exactly $2k$. □

From the lemma above we can deduce that there are binary morphisms that can greatly increase the number of BWT-runs of some words. We define the sensitivity of BWT-runs

$\mathtt{a}$	$\dots$	$\dots$	$\mathtt{b}$
$\vdots$	$\vdots$	$\vdots$	x
$\mathtt{a}$	$\mathtt{b}^k\mathtt{a}u_k^R$	$\dots$	$\mathtt{a}$
$\mathtt{ba}$	$\mathtt{a}^{2k}\mathtt{b}$	$\dots$	$\mathtt{a}$
$\vdots$	$\vdots$	$\vdots$	y
$\mathtt{ba}$	$\mathtt{bb}$	$\dots$	$\mathtt{a}$
$\mathtt{b}^2\mathtt{a}$	u_2	$\dots$	$\mathtt{a}$
$\mathtt{b}^2\mathtt{a}$	u_3	$\dots$	$\mathtt{ab}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\mathtt{b}^2\mathtt{a}$	u_k	$\dots$	$\mathtt{ab}^{k-2}$
$\mathtt{b}^2\mathtt{a}$	u_k^R	$\dots$	$\mathtt{ab}^{k-2}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\mathtt{b}^2\mathtt{a}$	u_3^R	$\dots$	$\mathtt{ab}$
$\mathtt{b}^2\mathtt{a}$	u_2^R	$\dots$	$\mathtt{a}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\mathtt{b}^k\mathtt{a}$	u_k	$\dots$	$\mathtt{a}$
$\mathtt{b}^k\mathtt{a}$	u_k^R	$\dots$	$\mathtt{a}$

$$\xRightarrow[\text{reverse order}]{\theta\equiv(\mathtt{ab},\mathtt{aa})}$$

$\mathtt{b}$	$\dots$	$\dots$	$\mathtt{a}$
$\mathtt{ab}$	$\dots$	$\dots$	$\mathtt{aa}$
$\vdots$	$\vdots$	$\vdots$	$\overline{x}$
$\mathtt{ab}$	$\theta(\mathtt{b}^k\mathtt{a}u_k^R)$	$\dots$	$\mathtt{ab}$
$(\mathtt{aa})\mathtt{b}$	$\dots$	$\dots$	$\mathtt{a}$
$(\mathtt{aa})\mathtt{ab}$	$\theta(\mathtt{a}^{2k}\mathtt{b})$	$\dots$	$\mathtt{ab}$
$\vdots$	$\vdots$	$\vdots$	$\overline{y}$
$(\mathtt{aa})\mathtt{ab}$	$\theta(\mathtt{bb})$	$\dots$	$\mathtt{ab}$
$(\mathtt{aa})^2\mathtt{b}$	$\dots$	$\dots$	$\mathtt{a}$
$(\mathtt{aa})^2\mathtt{ab}$	$\theta(u_2)$	$\dots$	$\mathtt{ab}$
$(\mathtt{aa})^2\mathtt{ab}$	$\theta(u_3)$	$\dots$	$\mathtt{aa}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$(\mathtt{aa})^2\mathtt{ab}$	$\theta(u_k)$	$\dots$	$\mathtt{aa}$
$(\mathtt{aa})^2\mathtt{ab}$	$\theta(u_k^R)$	$\dots$	$\mathtt{aa}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$(\mathtt{aa})^2\mathtt{ab}$	$\theta(u_3^R)$	$\dots$	$\mathtt{aa}$
$(\mathtt{aa})^2\mathtt{ab}$	$\theta(u_2^R)$	$\dots$	$\mathtt{ab}$
$(\mathtt{aa})^3\mathtt{b}$	$\dots$	$\dots$	$\mathtt{a}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$(\mathtt{aa})^k\mathtt{b}$	$\dots$	$\dots$	$\mathtt{a}$
$(\mathtt{aa})^k\mathtt{ab}$	$\theta(u_k)$	$\dots$	$\mathtt{ab}$
$(\mathtt{aa})^k\mathtt{ab}$	$\theta(u_k^R)$	$\dots$	$\mathtt{ab}$

Figure 5.5: To the left is the BWT matrix of z_k. To the right is the BWT matrix of $\theta(z_k)$, here displayed in reverse order. Each gray row represents a block of rotations from $\mathcal{S}_\theta(z_k)$ starting with the same prefix, highlighted in the first column. Each one of these block except the first one yields 2 extra runs on $\mathtt{bwt}(\theta(z_k))$. The words x and y correspond to the concatenation of the last letters of blocks of the BWT matrix of z_k whose form is unknown, but do not play a role in the increase on $r(\theta(z_k))$.

to morphism application in a similar way to how Akagi et al. define the sensitivity of repetitiveness measures to edit operations [1].

Definition 5.2.30 The *BWT additive sensitivity* and *BWT multiplicative sensitivity* for a morphism φ are respectively, the functions

$$AS_{r,\varphi}(n) = \max_{w\in\Sigma^n}(\Delta^+_\varphi(w)) \text{ and } MS_{r,\varphi}(n) = \max_{w\in\Sigma^n}(\Delta^\times_\varphi(w))$$

Proposition 5.2.31 Let θ be the period-doubling morphism. It holds that $AS_{r,\theta}(n) = \Omega(\sqrt{n})$.

Proof. The length of the words z_k in Lemma 5.2.29 is $n = \Theta(k^2)$. We showed that $\Delta^+_\theta(z_k) = 2k = \Theta(\sqrt{n})$ on these words. For values of n in between $|z_k|$ and $|z_{k+1}|$, it is easy to see that for the word $z_k\mathtt{a}^j$ for $0 < j < |z_{k+1}| - |z_k|$, it still holds that $\Delta^+_\theta(z_k\mathtt{a}^j) = 2k$, as none of the key aspects of the proof of Lemma 5.2.29 changes. Thus, the claim holds. □

5.2.6 Comparison with other repetitiveness measures

Morphisms behave very differently when other repetitiveness measures are considered. For instance, any morphism φ increases the size $z(w)$ of the Lempel-Ziv parsing of any word by at most an additive constant depending only on φ. This holds for any alphabet size, as shown by Constantinescu and Ilie [35, Lemma 8].

Lemma 5.2.32 Let $\varphi : \Sigma^* \to \Gamma^*$ be any morphism. For every word w, it holds that $z(\varphi(w)) \leq z(w) + k$ where k is a constant depending only on φ.

The result of Constantinescu and Ilie can easily be extended to the LZ parsing without overlaps, the optimal (not the greedy) LZ-end parsing, and bidirectional macro schemes. We can show a similar result for the size $g(w)$ of the smallest deterministic context-free grammar generating only w. This can be further generalized to the size of the smallest run-length context-free grammar, and also to the size of the smallest collage system.

Lemma 5.2.33 Let $\varphi : \Sigma^* \to \Gamma^*$ be any (possible erasing) morphism. For every word w, it holds that $g(\varphi(w)) \leq g(w) + k$ where k is a constant depending only on φ.

Proof. Given a deterministic context-free grammar G of size $|G|$ generating w, we construct a grammar generating $\varphi(w)$. For each occurrence of a terminal symbol a in any rule of the grammar, replace it with a new non-terminal A_a. For each terminal symbol add the rule $A_a \to \varphi(a)$. The size of the resulting grammar is $g' \leq |G| + k$ where $k = \sum_{a \in \Sigma} |\varphi(a)|$. Let G be the smallest grammar generating w, and then the thesis holds. If the resulting grammar has erasing rules, we can delete them, and replace the occurrences of those erasing variables in other variables by ε. We repeat this recursively. The size of the resulting grammar can only decrease, so the thesis still holds. □

If for some fixed measure and morphism, this morphism increases the value of the measure always by at most a fixed constant, then we can derive an easy upper bound for the family of words obtained by iterating that morphism.

Proposition 5.2.34 Let μ be a repetitiveness measure and φ be a morphism. Suppose that for every word w it holds that $\mu(\varphi(w)) \leq \mu(w) + k$ for a constant k depending only on μ and φ. Then, $\mu = \mathcal{O}(i)$ in the family $\{\varphi^i(w) \mid i \geq 0\}$.

Proof. Let $k' = \mu(\varphi(w))$. We show by induction that $\mu(\varphi^i(w)) \leq ki + k'$ for any $i \geq 1$. For $i = 1$, clearly $\mu(\varphi(w)) \leq k + k'$. Let $i > 1$ and suppose the claim is true for $i - 1$. Then, $\mu(\varphi^i(w)) \leq \mu(\varphi^{i-1}(w)) + k \leq (k(i-1) + k') + k \leq ki + k'$. □

The families on the proposition above are known as *D0L-sequences* [120]. As a direct consequence of Lemma 5.2.33 and Proposition 5.2.34, it holds that all repetitiveness measures upper-bounded by g are $\mathcal{O}(i)$ on the family of words belonging to a fixed D0L-sequence. In fact, the result we obtain is even more general because we can apply any morphism to words obtained from a D0L-sequence increasing the size of the grammar only by a fixed constant.

Proposition 5.2.35 For every (possibly erasing) morphisms φ and λ, and every word w, it holds that $g = \mathcal{O}(i)$ in the family $\{\lambda \circ \varphi^i(w) \,|\, i \geq 0\}$.

Proof. By Lemma 5.2.33 and Proposition 5.2.34, it holds that $g(\varphi^i(w)) = \mathcal{O}(i)$ for every (possibly erasing) morphism φ. By Lemma 5.2.33, one has $g(\lambda \circ \varphi^i(w)) \leq g(\varphi^i(w)) + k$ for every (possibly erasing) morphism λ, and a constant k depending on λ. Thus, $g(\lambda \circ \varphi^i(w)) = \mathcal{O}(i)$. □

It is unknown if an analogous result is true for r. In fact, even for the restricted case of purely morphic words, this is known to hold only for the binary case [47].

Chapter 6

New Repetitiveness Measures Based on Self-Similarity

In this chapter, we explore a source of repetitiveness that is more structural than the one captured by the measures δ, γ, and bidirectional macro schemes. It is captured by *string morphisms*, so we define new measures based on them. We call this source of repetitiveness *self-similarity.* We show that by exploiting this source, we can sharply break the lower bound for repetitiveness given by δ.

The chapter is structured as follows.

- In Section 6.1, we introduce our simplest mechanisms exploiting self-similarity, which we call *L-systems*, and build upon deterministic *Lindenmayer systems* [88, 89], in particular on the variant called *CPD0L-systems.* A CPD0L-system describes the language of the images, under a coding τ, of the powers of a non-erasing morphism φ starting from a string s (called the *axiom*), that is, the set $\{\tau(\varphi^i(s)) \mid i \geq 0\}$. L-systems extend CPD0L-systems with two parameters, d and n, so as to unambiguously describe the string $w = \tau(\varphi^d(s))[1\,..\,n]$. The size of the shortest description of an L-system generating w in this way is called $\ell(w)$. Intuitively, ℓ captures repetitiveness because any occurrence of a symbol a in $\varphi^i(s)$ expands to the same string in $\varphi^{i+j}(s)$ for every j. The resulting repetitiveness is, however, structured by the morphism φ, instead of completely free as in BMSs. We show how to perform basic operations like decompression or direct access on L-systems, and prove that they are monotone upon appending prefixes.
- In Section 6.2, we show that ℓ can be much smaller than δ, by up to a $\Theta(\sqrt{n})$ factor. We also show that ℓ can be $\Omega(\delta \log n)$ in other string families, which makes ℓ uncomparable to δ. This implies that the lower bound δ does not capture the same kind of repetitiveness. On the other hand, we show that $\ell = \mathcal{O}(g)$. This bound is important because it implies that δ can be only polylogarithmically smaller than ℓ, and places ℓ within the map of known repetitiveness measures.
- In Section 6.3, we expose string families where ℓ is larger than the output of several repetitiveness-aware compressors like $g_{\texttt{rl}}$, $z_{\texttt{e}}$, r and $r_{\$}$. We then conclude that ℓ is uncomparable to almost all measures other than g, which suggests that the source of

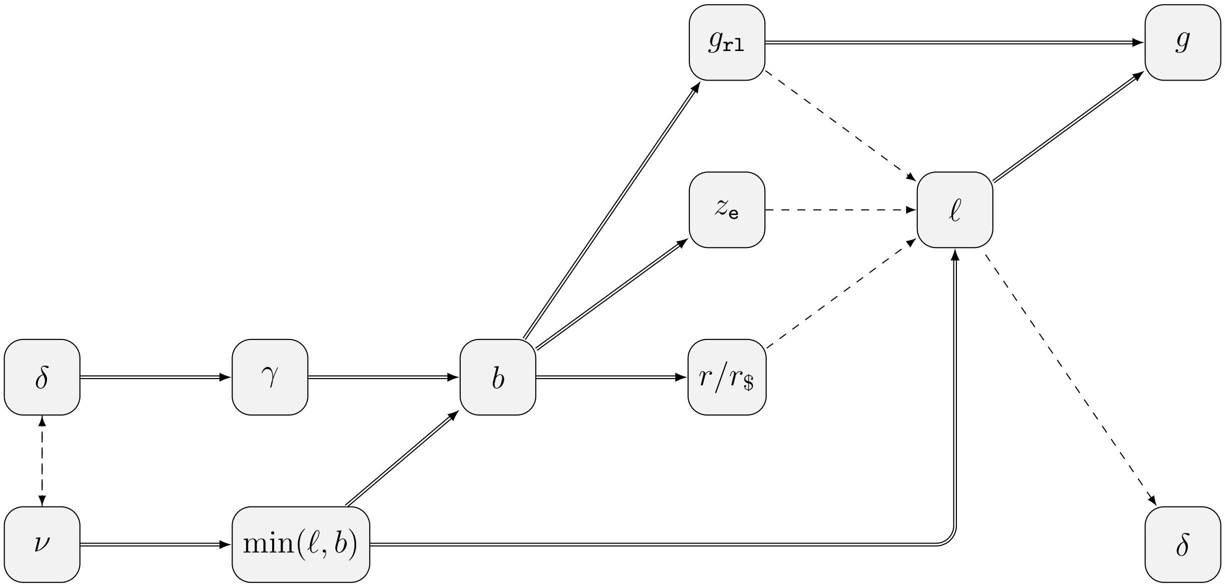


Figure 6.1: Asymptotic relations between ℓ, ν, and other repetitiveness measures. A double solid arrow from v_1 to v_2 means that it always holds that $v_1 = \mathcal{O}(v_2)$, and there exists a string family where $v_1 = o(v_2)$. A dashed arrow from v_1 to v_2 means that there exists a family where $v_1 = o(v_2)$. We suggest the reader to check Figure 4.3 for further implications.

repetitiveness it captures is largely orthogonal to the typical cut-and-paste of macro schemes.

- In Section 6.4, we introduce *macro-systems*, which are a reformulation of bidirectional macro schemes, in the sense that the size of the smallest (internal) macro system is $\Theta(b)$, where b is the size of the smallest BMS. This formulation makes them easy to combine with L-systems in the following section.
- In Section 6.5, we introduce NU-systems, which elegantly combine L-systems and BMSs, and the measure ν, defined as the size of the smallest NU-system generating the string. We introduce a string family where ν is asymptotically strictly smaller than both ℓ and b, which shows that NU-systems are indeed relevant and positions ν as the unique smallest reachable repetitiveness measure to date that captures both kinds of repetitiveness in non-trivial ways. We also study how to decompress NU-systems and its sensitivity to some operations on the string.
- In Section 6.6, we study various ways of simplifying L-systems and show that, in all the cases we considered, we end up with a weaker repetitiveness measure. We also show that some of those weaker variants of ℓ can be of independent interest, as they speed up some relevant processes like decompression and direct access.

Overall, our results contribute to understanding how to measure repetitiveness and how to exploit it in order to build better compressors. Figure 6.1 shows how our new measures ℓ and ν relate to other measures in the literature.

6.1 Deterministic L-systems and the Measure ℓ

In this section we study a mechanism for generating infinite sequences called *deterministic Lindenmayer systems* (L-systems) [88, 89], which build on string morphisms. L-systems were initially utilized as a tool to model the growth of plants and algae [88, 89]. They also have been used to define infinite words with interesting self-similarity and factor complexity properties [120]. For these reasons, L-systems have been studied extensively from a practical and mathematical point of view. We adapt L-systems to generate finite repetitive strings. L-systems are, in essence, grammars with only non-terminals, which typically generate longer and longer strings, in a levelwise fashion. For our purposes, we will also specify at which level d to stop the generation process and the length n of the string w to generate. The generated string $w[1\mathinner{.\,.}n]$ is then the prefix of length n of the sequence of variables obtained at level d.

We adapt, in particular, the variant called *CPD0L-systems*, though we will use the generic name *L-systems* for simplicity. Formally, a *CPD0L-system* is a 4-tuple $L = (\Sigma, \varphi, \tau, s)$, where Σ is the *alphabet*, φ is the set of *rules* (a non-erasing endomorphism on Σ^*), τ is a coding on Σ^*, and $s \in \Sigma$ is the *initial symbol* or the *axiom*. The system generates the sequence $(\tau(\varphi^d(s)))_{d\in\mathbb{N}}$. The “D0L” stands for *deterministic L-system with 0 interactions*, which means that the L-system has one rule per symbol and that rules are context-free. The “P” stands for *propagating*, which means that φ is non-erasing. Finally, the “C” stands for *coding*, which means that the system is extended with a coding. To define a compressor based on CPD0L-systems, we extend them to 6-tuples by fixing d and using another parameter n, so we can uniquely determine a string of the sequence generated by the system and then extract a prefix from it.

Definition 6.1.1 (L-systems) An *L-system* is a 6-tuple $L = (\Sigma, \varphi, \tau, s, d, n)$ where Σ is the *alphabet*, φ is the set of *rules* (a non-erasing endomorphism on Σ^*), τ is a coding on Σ^*, $s \in \Sigma$ is the *axiom*, and d and n are two non-negative integers. The string generated by L is $w = \tau(\varphi^d(s))[1\mathinner{.\,.}n]$.

We now define the size of an L-system and the measure ℓ.

Definition 6.1.2 (Measure ℓ) The *size* of an L-system $L = (\Sigma, \varphi, \tau, s, d, n)$ is $\texttt{size}(L) = \texttt{size}(\varphi) + |\Sigma| + 3$. The measure $\ell(w)$ is defined as the size of the smallest L-system generating w.

The size of an L-system accounts for the lengths of the right-hand sides of the rules in φ, the coding τ, the axiom symbol, and the values d and n, so we can effectively represent the system using $\mathcal{O}(\texttt{size}(L))$ space. Hence, the measure ℓ is reachable. As a convention, we always assume that d and $|\Sigma|$ are in $n^{\mathcal{O}(1)}$. Otherwise, we would need $\omega(1)$ $\Theta(\log n)$-bit words to represent the integer d or the symbols of the alphabet. We also assume that $\texttt{size}(L) \le 3n$ (and hence $|\Sigma| \le 3n$), as there is always a trivial L-system $L = (\Sigma', \{s \to w\}, \texttt{id}, s, 1, n)$ generating w, of size $n + |\Sigma'| + 3$, where Σ' contains the symbols actually appearing in w, and $\texttt{id}$ is the identity function. A finer-grained analysis of the number of bits needed to represent an L-system of size ℓ yields $\mathcal{O}(\ell \log |\Sigma| + \log n)$ bits, the second term corresponding to d and n; note Σ contains the alphabet of w.

Algorithm 2 Decompressing L-system $L = (\Sigma, \varphi, \tau, s, d, n)$ in time $\mathcal{O}(dn)$; invoke with DECOMPRESS(s, d, n).

```
Input  : Symbol $a$ to expand, number of levels $d$, maximum length to output $n > 0$.
Output: The string $\tau(\varphi^d(a))[1\,..\,n']$ with $n' = \min(n, |\varphi^d(a)|)$. Returns $n - n'$.
function DECOMPRESS$(a, d, n)$
    if $d = 0$ then
        output $\tau(a)$
        return $n - 1$
    let $a \to b_1 \cdots b_k \in \varphi$
    for $i \leftarrow 1$ to $k$ do
        $n \leftarrow$ DECOMPRESS$(b_i, d - 1, n)$
        if $n = 0$ then return 0
    return $n$
```

6.1.1 Decompression

In this subsection we design and analyze algorithms for decompressing L-systems. The decompression of L-systems is, in principle, very similar to that of context-free grammars, except that we must keep track of the level so as to output $\tau(a)$ when we reach a symbol a at level d. We must also keep track of the number of symbols already output so as to stop when they reach n.

This simple procedure, depicted in Algorithm 2, takes time $\mathcal{O}(dn)$; consider the example system $L = (\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, \{\mathtt{a} \to \mathtt{ab}, \mathtt{b} \to \mathtt{c}, \mathtt{c} \to \mathtt{b}\}, \tau, \mathtt{a}, d, n)$. The root of this inefficiency is the cycle $\mathtt{b} \leftrightarrow \mathtt{c}$, which allows the string not to grow with d. Removing "unary" symbols, that is, with right-hand side of length 1, is not as simple as with CFGs, but it is possible and yields better decompression time.

To properly eliminate unary symbols, we define the function $f : \Sigma \to \Sigma$ such that $f(a) = b$ iff the rule for a starts with b, $a \to b \cdots$. In our example, $f(\mathtt{a}) = \mathtt{a}$, $f(\mathtt{b}) = \mathtt{c}$, and $f(\mathtt{c}) = \mathtt{b}$. A representation of function f can be built in $\mathcal{O}(|\Sigma|)$ time and space so that $f^h(a)$ can be computed in constant time for any $h \geq 0$ [98]. We also define function g as $g(a) = \min\big(\{h \geq 0 \mid f^h(a) \text{ is not a unary symbol}\} \cup \{+\infty\}\big)$. In our example, $g(\mathtt{a}) = 0$ and $g(\mathtt{b}) = g(\mathtt{c}) = +\infty$. It is an easy exercise to build function g in time $\mathcal{O}(|\Sigma|)$, by trying, for each a not already visited, a, $f(a)$, $f^2(a)$, $\ldots$ until finding the first non-unary symbol $f^h(a)$, and then filling $g(f^k(a)) = h - k$ for all $0 \leq k \leq h$, or $+\infty$ for all of them if we fall in a loop of unary symbols.

Algorithm 3 shows the improved procedure. Every unary path in the derivation of the output is now traversed in constant time. The nodes of the recursion tree then have at least two children, except for those on the rightmost path, which may have only one child included in the prefix of length n. Since the recursion tree has n leaves and depth d, it has $\mathcal{O}(n + d)$ nodes, the term n counting the leaves and their non-unary ancestors, and d counting those rightmost nodes that are possibly unary. The bound is tight; consider our example L-system with small n and large d. The total decompression time is then $\mathcal{O}(|\Sigma| + n + d)$, where $\mathcal{O}(|\Sigma|)$ counts the time to build f and g. Recall we can assume $|\Sigma| = \mathcal{O}(n)$.

Algorithm 3 Decompressing L-systems $L = (\Sigma, \varphi, \tau, s, d, n)$ in time $\mathcal{O}(n + d)$; invoke with `decompress`(s, d, n).

```
Input  : Symbol $a$ to expand, number of levels $d$, maximum length to output $n > 0$.
Output: The string $\tau(\varphi^d(a))[1\,..\,n']$ with $n' = \min(n, |\varphi^d(a)|)$. Returns $n - n'$.
function DECOMPRESS$(a, d, n)$
    $h \leftarrow g(a)$
    if $h \geq d$ then
        output $\tau(f^d(a))$
        return $n - 1$
    $b \leftarrow f^h(a)$
    let $b \rightarrow b_1 \cdots b_k \in \varphi$
    for $i \leftarrow 1$ to $k$ do
        $n \leftarrow$ DECOMPRESS$(b_i, d - h - 1, n)$
        if $n = 0$ then return $0$
    return $n$
```

Theorem 6.1.3 Given an L-system $L = (\Sigma, \varphi, \tau, s, d, n)$, we can compute its represented string in time $\mathcal{O}(n + d)$.

In case d is significantly larger than n, the following solution that decompresses in time $\mathcal{O}(|\Sigma| + n \log d)$ may be of interest. With functions f and g, we follow a procedure similar to one used on CFGs [54] for decompressing in real time: To decompress a symbol a with d levels, we first output $f^d(a)$. Now let $b = f^{d-1}(a)$ and $b \rightarrow a\, b_2 \cdots b_k$. We recursively decompress $b_2, \ldots, b_k$ with $d = 0$. Now let $c = f^{d-2}(a)$ and $c \rightarrow b\, c_2 \cdots c_r$. We recursively decompress $c_2, \ldots, c_r$ with $d = 1$, and so on. The procedure finishes when we have output n symbols or we have completely expanded a with d levels.

This algorithm outputs a symbol per unit of work done, except when we try $c = f^h(a)$ for some $h = d - 1, \ldots, 0$ and c is unary. Those unary symbols that we visit as we return from level d take one unit of work and yield no symbols. To avoid wasting time on them, we use function g. Instead of trying out all the values of h from $d - 1$ to 0, we use binary search to skip the unary nodes; see Algorithm 4. The binary search is possible because, if $g(f^{d/2}(a)) \geq d/2$, then the largest $0 \leq h < d$ with a non-unary symbol is in $[0\,..\,d/2 - 1]$, otherwise it is in $[d/2\,..\,d - 1]$. In the worst case, this poses a penalty of $\mathcal{O}(\log d)$ to every symbol output. The bound is tight even if we use doubling search; consider the L-system $L = (\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, \{\mathtt{a} \rightarrow \mathtt{ba}, \mathtt{b} \rightarrow \mathtt{c}, \mathtt{c} \rightarrow \mathtt{b}\}, \tau, \mathtt{a}, d, n)$ with large d.

Theorem 6.1.4 Given an L-system $L = (\Sigma, \varphi, \tau, s, d, n)$, we can compute its represented string in time $\mathcal{O}(n \log d)$.

6.1.2 Access

A more ambitious goal in compression formats is to provide *direct access* to the represented string w, that is, being able to retrieve $w[i\,..\,j]$ without the need to decompress the whole w. This can be done in time $\mathcal{O}(j - i + \log n)$ on grammar-based representations [18], but it is

Algorithm 4 Decompressing L-systems $L = (\Sigma, \varphi, \tau, s, d, n)$ in time $\mathcal{O}(n \log d)$; invoke with $\textsc{decompress}(s, d, n)$.

```
Input   : Symbol $a$ to expand, number of levels $d$, maximum length to output $n > 0$.
Output: The string $\tau(\varphi^d(a))[1..n']$ with $n' = \min(n, |\varphi^d(a)|)$. Returns $n - n'$.
function $\textsc{decompress}(a, d, n)$
    output $\tau(f^d(a))$
    $n \leftarrow n - 1$
    if $n = 0$ then return $0$
    $h \leftarrow d - 1$
    while $h \geq 0$ do
        $b \leftarrow f^h(a)$
        let $b \rightarrow b_1 \cdots b_k \in \varphi$
        for $j \leftarrow 2$ to $k$ do
            $n \leftarrow \textsc{decompress}(b_j, d - h - 1, n)$
            if $n = 0$ then return $0$
    $h \leftarrow \max\big(\{k \in [0..h{-}1] \mid k + g(f^k(a)) < h\} \cup \{-1\}\big)$ (binsearch)
    return $n$
```

not known if it can be done on bidirectional macro schemes or even Lempel-Ziv variants.

We first focus on accessing the single symbol $w[i]$. For any $a \in \Sigma$, we define $a^l = \varphi^l(a)$ as the string obtained by iterating l times the morphism φ on a. Now let $a \rightarrow b_1 \cdots b_k \in \varphi$, then we define $p_a^l(t) = \sum_{r=1}^{t} |b_r^l|$ for $0 \leq t \leq k$.

We begin the extraction of $w[i]$ from the axiom $a_0 = s$, with $i_0 = i$. Let $r_0 \geq 1$ be such that $p_{a_0}^d(r_0 - 1) < i_0 \leq p_{a_0}^d(r_0)$, and $a_0 \rightarrow b_1 \cdots b_k \in \varphi$. Then $w[i] = \tau(a_1^{d-1}[i_1])$, with $a_1 = b_{r_0}$ and $i_1 = i_0 - p_{a_0}^d(r_0 - 1)$. After continuing for d levels, we finally have $w[i] = \tau(a_d)$. Algorithm 5 shows the process.

With binary search, the algorithm takes time $\mathcal{O}(d \log |\varphi|)$. We can improve it by using instead interval-biased search trees [18] on the sequences p_a^l. With those trees, the search for x on a sequence of values $i_1 < \cdots < i_t$ within a universe of size u takes time $\mathcal{O}(\log(u/(i_{r+1} - i_r)))$, if $i_r < x \leq i_{r+1}$. By pruning the values of the sequences p_a^l to a maximum of n, we have that the first search, on s, will take time $\mathcal{O}(\log(n/|a_1^{d-1}|))$, the second one $\mathcal{O}(\log(|a_1^{d-1}|/|a_2^{d-2}|))$, and so on, which telescopes to $\mathcal{O}(d + \log n)$. Note that $n \leq |\varphi|^d$, so $d + \log n = \mathcal{O}(d \log |\varphi|)$ and it could be less.

Let us now consider how to preprocess the L-system to compute p_a^l. We define the $|V| \times |V|$ matrix M_φ, so that $M_\varphi[a][b]$ is the number of times b appears in the right-hand side of the rule for a (cf. [123]). Formally, if $a \rightarrow b_1 \cdots b_k \in \varphi$, then $M_\varphi[a][b] = |\{r \mid b_r = b\}|$. Now note that the vector $L_1 = M_\varphi \times [1 \cdots 1]^T$ is such that $L_1[a] = |a^1|$, and in general, $L_l = M_\varphi^l \times [1 \cdots 1]^T$ satisfies $L_l[a] = |a^l|$. Since M_φ contains only $|\varphi|$ nonzero entries, we can compute all the vectors L_l by defining $L_0 = [1 \cdots 1]^T$ and each $L_l = M_\varphi \times L_{l-1}$ for $l = 1, \ldots, d$, in $\mathcal{O}(d(|\varphi| + |V|)) = \mathcal{O}(d|\varphi|)$ total time and $\mathcal{O}(|\varphi| + d|V|)$ space. Once the vectors L_l are obtained, we can compute the functions p_a^l in $\mathcal{O}(d|\varphi|)$ space and time. The interval-biased search trees are built in linear time and space, which adds up to $\mathcal{O}(d|\varphi|)$ in our case.

Algorithm 5 Accessing $w[i]$ from L-system $L = (\Sigma, \varphi, \tau, s, d, n)$ in time $\mathcal{O}(d + \log n)$; invoke with ACCESS(s, d, i).

```
Input  : Axiom $s$ to expand, number of levels $d$, position to access $1 \leq i \leq n$.
Output: The symbol $\tau(\varphi^d(s)[i])$.
1: function ACCESS$(s, d, i)$
2:     let $p_a^l$ be precomputed for all $a \in \Sigma$, $l \in [0 \,.\,. d-1]$
3:     $a \leftarrow s$
4:     for $l \leftarrow d$ to 1 do
5:         let $a \rightarrow b_1 \cdots b_k \in \varphi$
6:         let $r$ be such that $p_a^l(r-1) < i \leq p_a^l(r)$ (interval-biased search)
7:         $i \leftarrow i - p_a^l(r-1)$
8:         $a \leftarrow b_r$
9:     return $\tau(a)$
```

Theorem 6.1.5 After an $\mathcal{O}(d|\varphi|)$ space and time preprocessing of an L-system $L = (\Sigma, \varphi, \tau, s, d, n)$ representing w, we can extract any substring $w[i \,.\,. j]$ in time $\mathcal{O}(j - i + d + \log n)$.

PROOF. We have already described the preprocessing and how to access an individual cell. Assume we access $w[i]$ and $w[j]$. Their paths along Algorithm 5 may coincide for some levels, until they diverge on the right-hand side of some rule $a \rightarrow b_1 \cdots b_k$ at some level l. From levels $t = l \,.\,. d$, the access to $w[i]$ computes values $r_t = r$ from the right-hand sides of the rules of a_t in line 6. Similarly, the access to $w[j]$ computes values r'_t and a'_t. We then output the following strings, in this order, which form $w[i \,.\,. j]$:

1. $\tau(\varphi^{d-t}(b_{r_t})), \tau(\varphi^{d-t}(b_{r_t+1})), \ldots$, with $a_t \rightarrow b_1 \cdots$, for $t = d, d-1, \ldots, l+1$.
2. $\tau(\varphi^{d-l}(b_{r_l+1}))$, $\ldots$, $\tau(\varphi^{d-l}(b_{r'_l-1}))$.
3. $\tau(\varphi^{d-t}(b_1)), \ldots, \tau(\varphi^{d-t}(b_{r'_t-1}))$, with $a'_t \rightarrow b_1 \cdots$, for $t = l+1, l+2, \ldots, d$.

Each of those whole subtrees, say for symbols $b^{l'}$, are decompressed in optimal time using function `decompress`$(b, l', |b^{l'}| = L_{l'}[b])$ from Algorithm 3. Since the algorithm decompress the whole symbol, its recursion tree has $|b^{l'}|$ leaves and then it has maximum height $|b^{l'}|$; therefore it runs in optimal time $\mathcal{O}(|b^{l'}|)$. □

The extraction time $\mathcal{O}(j - i + \log n)$ is near-optimal for any representation of size $\mathcal{O}(g)$ [128], which we show to be the case of L-systems in Section 6.2. The extra term $\mathcal{O}(d)$, which is related to the height of the grammar, has been removed (or reduced to $\mathcal{O}(\log n)$) on grammars via heavy-path decomposition of the parse tree [18] or by balancing the grammar [52]. Applying either technique to L-systems is an interesting challenge. Another relevant challenge is to decrease our heavy preprocessing space and time of $\mathcal{O}(d|\varphi|)$, even if we can perform it once and then answer many extraction queries.

In this sense, the closest result to ours, by Salomaa [123], precomputes the characteristic function of the matrix M_φ, which allows computing $|\varphi^d(s)|$ in time $\mathcal{O}(|\Sigma| \log d)$ (the function has $|\Sigma|$ terms that include polynomials and exponents on d). By precomputing the function for every possible initial symbol $a \in \Sigma$, we use $\mathcal{O}(|\Sigma|^2)$ space (which typically compares favorably to our space $\mathcal{O}(d|\varphi|)$) and can compute any value p_a^l in time $\mathcal{O}(|\Sigma| \log d)$; this yields an extraction time of $\mathcal{O}(j - i + (d + \log n)|\Sigma| \log n)$. The precomputation time for the $|\Sigma|$ symbols is $\mathcal{O}(|\Sigma|^{3.7})$ arithmetic operations, dominated by the time to find the characteristic function on integer matrices [64]. Shallit and Swart [125] aim to remove the $\mathcal{O}(d)$ term from the extraction time, by using the cycles in the grammar in order to jump near the desired level. They manage to compute any $\varphi^d(a)[i]$ in time bounded by a polynomial (yet of degree 10) in $|\Sigma|, \texttt{width}(\varphi), \log d$ and $\log i$.

6.1.3 Sensitivity to string transformations

When considering repetitiveness measures, it is often useful to know how they can change after applying relevant string transformations on the input. In the case of the measure ℓ, we can show that this measure is monotone with respect to prepending symbols.

Proposition 6.1.6 Let $w \in \Sigma^+$ and a symbol a. It holds that $\ell(aw) \leq \ell(w) + \mathcal{O}(1)$.

Proof. Assume first that $a \in \Sigma$. Let $L = (\Sigma, \varphi, \tau, s, d, n)$ be a minimal L-system representing w, and two new distinct symbols $b, s' \notin \Sigma$. Let the L-system $L' = (\Sigma', \varphi', \tau', s', d', n')$, with $\Sigma' = \Sigma \cup \{b, s'\}$, $\varphi = \varphi \cup \{b \to b, s' \to bs\}$, $\tau' = \tau \cup \{b \to a, s' \to s'\}$, $d' = d + 1$, and $n' = n + 1$. Clearly L' generates aw and its size is $\texttt{size}(L) + 5$. Thus, $\ell(aw) \leq \ell(w) + 5$. If $a \notin \Sigma$ then we just let $b = a$ and the claim follows. □

On the other hand, basic edit operations like the insertion, deletion, or substitution of a single symbol at an arbitrary position (or even just at the end of the string) are not straightforward to handle. More complex operations like reversing w, or applying to it a string morphism different from φ, are also non-trivial to analyze.

6.2 Breaking the Repetitiveness Lower Bound δ

The measure δ is a (strict) lower bound to all the other usually considered repetitiveness measures [79, 99]. It is also a lower bound to the k-th order empirical entropy, which is a lower bound for statistical compression [99]. This implies that δ is an asymptotic lower bound to the size of almost every existing compressor and compressibility measure to date.

Since δ is unreachable in general [79], we cannot expect to find a reachable measure smaller than δ. We are interested, instead, in reachable measures that also capture repetitiveness and go below δ in some restricted but relevant scenarios. While it is always possible to design a measure that breaks δ on some specific string families, we require this measure to be *competitive*, meaning that it is at least as good as other better established measures like z, g, or $r/r_\$$.

As we show next, the repetitiveness measure ℓ satisfies those conditions. Indeed, we show that δ and ℓ are uncomparable. We first show that ℓ can be larger than δ by a logarithmic factor.

Lemma 6.2.1 There exist string families where $\ell = \Omega(\delta \log n)$.

Proof. Kociumaka et al. [79] exhibit a string family of $2^{\Theta(\log^2 n)}$ elements with $\delta = \mathcal{O}(1)$, so it needs $\Omega(\log^2 n) = \Omega(\delta \log^2 n)$ bits to be represented with any method. On the other hand, an L-system of size ℓ is described with (at most) $\mathcal{O}(\ell \log n)$ bits. Therefore $\ell = \Omega(\log n) = \Omega(\delta \log n)$ in this family. □

On the other hand, ℓ is a competitive repetitiveness measure: the smallest L-system for a string is always asymptotically smaller than the smallest grammar. This shows that the measure ℓ is always reasonable for repetitive strings.

Lemma 6.2.2 It always holds that $\ell = \mathcal{O}(g)$.

Proof. Consider a grammar $G = (V, \Sigma, R, S)$ of height h generating $w[1\,..\,n]$. If there are rules $A \to \varepsilon$, remove them and remove A from all right-hand sides, iterating until all those rules disappear. We now define the equivalent L-system $L = (V \cup \Sigma, R', \tau, S, h, n)$, where R' contains all the rules in R plus the rules $a \to a$ for all $a \in \Sigma$. The coding is set to $\tau = \texttt{id}$.

It is clear that this L-system produces the same derivation tree of G, reaching terminals a at some level. Those remain intact up to the last level, h, thanks to the rules $a \to a$. At this point the L-system has derived $w[1\,..\,n]$.

The size of the L-system is that of G plus $\mathcal{O}(|\Sigma|)$, which is of the same order of the size of G because every symbol $a \in \Sigma$ appears on the right-hand side of some rule (if not, we can remove a from Σ). □

We now exhibit a string family where δ is $\Theta(\sqrt{n})$ times bigger than the smallest L-system. That is, ℓ can perform much better than δ in some scenarios.

Lemma 6.2.3 There exists a string family where $\delta = \Theta(\ell\sqrt{n})$.

Proof. Consider an L-system $L_d = (\Sigma, \varphi, \tau, s, d, n)$, where

$$\begin{aligned}
\Sigma &= \{\texttt{a}, \texttt{b}, \texttt{c}\} \\
\varphi &= \{\texttt{a} \to \texttt{a}, \texttt{b} \to \texttt{ab}, \texttt{c} \to \texttt{cb}\} \\
\tau &= \texttt{id} \\
s &= \texttt{c} \\
n &= 1 + \frac{d(d+1)}{2}
\end{aligned}$$

$$s_3 = \texttt{c b a b a a b}$$

$$s_6 = \texttt{c b a b a a } \underline{\texttt{b a a a b a}} \texttt{ a a a b a a a a a b}$$

Figure 6.2: All the substrings of length 6 of the string s_6 of Lemma 6.2.3 starting inside some position $i \leq |s_3| = 7$ are distinct, because the runs of $\texttt{a}$'s considered have all different and increasing lengths, and d is big enough. The last of the substrings considered is underlined. Extending these substrings one position to the left yields $|s_3|$ different strings of length 7, so the claim holds for even and odd values of $d \geq 2$.

for any $d \geq 0$. By iterating the morphism φ we obtain the words $s_d = \varphi^d(s)$:

$$\begin{aligned}
\varphi^0(\texttt{c}) &= \texttt{c}\\
\varphi^1(\texttt{c}) &= \texttt{cb}\\
\varphi^2(\texttt{c}) &= \texttt{cbab}\\
\varphi^3(\texttt{c}) &= \texttt{cbabaab}\\
\varphi^4(\texttt{c}) &= \texttt{cbabaabaaab}\\
\varphi^5(\texttt{c}) &= \texttt{cbabaabaaabaaaab}
\end{aligned}$$

and so on, from which we extract as a prefix the whole word. It is easy to check by induction that, for each $d \geq 0$, the string generated by the system L_d is $s_d = \texttt{c}\,\Pi_{i=0}^{d-1}\texttt{a}^i\texttt{b}$, which has length $1 + \frac{d(d+1)}{2}$.

It holds that ℓ is $\Theta(1)$ in this family: the system is essentially the same for every string in the family; the only changes are the integers d and n, which always fit in constant space.

On the other hand, the first $|s_{\lfloor d/2 \rfloor}| = 1 + \lfloor d/2 \rfloor(\lfloor d/2 \rfloor + 1)/2$ substrings of length d of s_d (for $d \geq 2$) are completely determined by the $\texttt{b}$'s they cross, and the number of $\texttt{a}$'s at their extremes, so they are all distinct. An example can be seen in Figure 6.2.

This gives the lower bound $\delta = \Omega(d) = \Omega(\sqrt{n})$. The upper bound $\mathcal{O}(\sqrt{n})$ holds trivially for run-length grammars, as the strings considered have $\Theta(\sqrt{n})$ runs of $\texttt{a}$'s followed by $\texttt{b}$'s, so $\delta = \mathcal{O}(g_{\texttt{rl}}) = \mathcal{O}(\sqrt{n})$. Therefore, it holds $\delta = \Theta(\sqrt{n}) = \Theta(\ell\sqrt{n})$ in this string family. □

The strings of Lemma 6.2.3 are easy to describe, yet hard to represent with copy-paste mechanisms. Intuitively, the simplicity of the sequence relies on the fact that many substrings can be structurally described in terms of previous ones, so it is arguably highly repetitive, though not via copy-paste. The repetitiveness in this family is better captured by an L-system, instead.

As a corollary of Lemmas 6.2.1 and 6.2.3, we obtain that ℓ and δ are uncomparable as repetitiveness measures.

Corollary 6.2.4 The measures ℓ and δ are uncomparable.

6.3 Uncomparability of ℓ with Other Measures

Given the uncomparability of ℓ and δ, a natural question is which other measures are also uncomparable to ℓ. We show in this section that this holds for almost every other repetitiveness measure. To do so, we first recall the string family of Kociumaka et al. [79], which needs $\Omega(\log^2 n)$ bits to be represented with any method. This string family will be crucial in the following proofs.

Definition 6.3.1 ([79]) The string family $\mathcal{K}$ is formed by all the infinite strings $\mathbf{s}$ over $\{\mathtt{a}, \mathtt{b}\}$ constructed as follows:

1. Let $\mathbf{s}[1] = \mathtt{b}$.
2. For any $i \geq 2$, choose a position j_i in $[2 \cdot 4^{i-2} + 1 \mathinner{.\,.} 4^{i-1}]$ and set $\mathbf{s}[j_i] = \mathtt{b}$.
3. If $j > 1$ and $j \neq j_i$ for any $i \geq 2$, let $\mathbf{s}[j] = \mathtt{a}$.

The family $\mathcal{K}_n$ for $n \geq 0$ is formed by all the prefixes of length n of some string in $\mathcal{K}$.

It is easy to see that the strings in the family $\mathcal{K}_n$ have $\Theta(\log n)$ symbol $\mathtt{b}$'s. Also, note that with the possible exception of the first two positions, there are no consecutive $\mathtt{b}$'s.

We are now ready to prove that, in general, it does not hold that $\ell = \mathcal{O}(g_{\mathtt{rl}})$, making L-systems uncomparable to RLCFGs.

Lemma 6.3.2 There exists a string family where $\ell = \Omega(g_{\mathtt{rl}} \log n / \log\log n)$.

Proof. Consider the string family $\mathcal{K}_n$ needing $\Omega(\log^2 n)$ bits (or $\Omega(\log n)$ space) to be represented with any method [79]. Strings in $\mathcal{K}_n$ have $\mathcal{O}(\log n)$ runs of $\mathtt{a}$'s separated by $\mathtt{b}$'s, so it is easy to see that $g_{\mathtt{rl}} = \mathcal{O}(\log n)$ in this family. Because of this, and because $g_{\mathtt{rl}}$ is a reachable measure, it holds that $g_{\mathtt{rl}} = \Theta(\log n)$ in $\mathcal{K}_n$. On the other hand, the minimal L-system for a string in this family can be represented with $\mathcal{O}(\ell \log|\Sigma| + \log n) \subseteq \mathcal{O}(\ell \log \ell + \log n)$ bits, which must be in $\Omega(\log^2 n)$ bits because the L-system is also reachable. It follows that $\ell = \Omega(\log^2 n / \log\log n)$, since otherwise

$$\ell \log \ell = o((\log^2 n / \log\log n) \log(\log^2 n / \log\log n)) = o(\log^2 n),$$

which contradicts ℓ being reachable. Therefore, in this string family it holds that $\ell = \Omega(g_{\mathtt{rl}} \log n / \log\log n)$. □

The same result holds for LZ-like parsings. Even the greedy LZ-End parsing (the largest of them) can be asymptotically smaller than ℓ in some string families.

Lemma 6.3.3 There exists a string family where $\ell = \Omega(z_{\mathtt{e}} \log n / \log\log n)$.

Proof. Take each string in $\mathcal{K}_n$ and prepend $\mathtt{a}^n$ to it. This new family of strings still needs $\Omega(\log^2 n)$ bits to be represented with any method because the size of the family is the same, and n just doubled. Just as in Lemma 6.3.2, it holds that $\ell = \Omega(\log^2 n / \log\log n)$ in this family. On the other hand, the LZ-End parsing needs $\Theta(\log n)$ phrases only to represent the prefix $\mathtt{a}^n\mathtt{b}$, and then for each run of $\mathtt{a}$'s followed by $\mathtt{b}$, its source is aligned with $\mathtt{a}^n\mathtt{b}$, so $z_{\mathtt{e}} = \Theta(\log n)$. Thus, $\ell = \Omega(z_{\mathtt{e}} \log n / \log\log n)$. □

The same result also holds for the number of equal-letter runs of the Burrows-Wheeler transform of a string.

Lemma 6.3.4 There exists a string family where $\ell = \Omega(r \log n / \log\log n)$.

Proof. Consider the family $\mathcal{K}_n$ again. Clearly $r = \Omega(\log n)$, because r is reachable. Because a string in this family has $\mathcal{O}(\log n)$ $\mathtt{b}$'s, its BWT has also $\mathcal{O}(\log n)$ runs of $\mathtt{a}$'s separated by $\mathtt{b}$'s (or the unique $\mathtt{\$}$, if included). Therefore, it holds that $r = \Theta(\log n)$ and $\ell = \Omega(r \log n / \log\log n)$ in this string family. □

We conclude that the measure ℓ is uncomparable to almost every other repetitiveness measure. We summarize these results in the following theorem.

Theorem 6.3.5 The measure ℓ is uncomparable to the repetitiveness measures δ, γ, b, v, c, $g_{\mathtt{rl}}$, z, $z_{\mathtt{no}}$, $z_{\mathtt{end}}$, $z_{\mathtt{e}}$, r and $r_{\mathtt{\$}}$.

Proof. There exist string families where $\ell = o(\delta)$. In these families, it holds $\ell = o(\mu)$ where μ is any of the measures considered above, because δ lower bounds them all. On the other hand, all the measures above are upper-bounded by at least one of $z_{\mathtt{e}}, g_{\mathtt{rl}}$, or r, which by Lemmas 6.3.2, 6.3.3, and 6.3.4, respectively, can be asymptotically smaller than ℓ for some string families. □

This shows that ℓ, although reachable and competitive, captures the regularities in strings in a form that is largely orthogonal to other repetitiveness measures.

6.4 Macro-systems

In this section we give a first step in combining L-systems with bidirectional macro schemes (BMSs) [127], by redefining BMSs in a way that makes them compatible with L-systems. This will allow us combining them in a straightforward manner. In our way, we obtain a generalization of BMSs. In Section 6.5 we combine macro-systems with L-systems, showing that mixing morphisms and copy-paste is more powerful than the sum of its parts.

We use the following formalism for BMSs.

Definition 6.4.1 A *bidirectional macro scheme (BMS)* for $w[1\,..\,n]$ is a *parse* $(x_1, s_1), \ldots, (x_b, s_b)$ of non-empty *phrases*, where $w = x_1 \cdots x_b$ and the second component is as follows: if $x_i = a$ is a single symbol, then it will be represented explicitly and $s_i = \perp$;

otherwise s_i is a position in w such that $w[s_i \mathinner{..} s_i + |x_i| - 1] = x_i$, indicating where we can copy x_i from. The BMS takes $\mathcal{O}(b)$ space, by representing the pairs (x_i, s_i) implicitly as $(|x_i|, s_i)$ if $|x_i| > 1$, and explicitly as $(x_i, \perp)$ if $|x_i| = 1$. We say that the *size* of the BMS is b.

To decompress a BMS, we define the function $\phi(j)$ that tells where to copy $w[j]$ from: let $e_t = \sum_{i=1}^{t} |x_i|$ and let p be such that $e_{p-1} < j \leq e_p$, that is, j belongs to the component (x_p, s_p) of the parse. Then, $\phi(j) = \perp$ if $|x_p| = 1$ (an explicit symbol) and otherwise $\phi(j) = (s_p - 1) + (j - e_{p-1})$.

A BMS is *valid* if for each j there exists $k \geq 0$ such that $\phi^k(j) = \perp$, and thus $w[j] = w[\phi^{k-1}(j)]$ if $k > 0$, and an explicit symbol if $k = 0$. We can then obtain $w[1 \mathinner{..} n]$ in $\mathcal{O}(n)$ time by:

1. Marking every cell as unknown, $w[j] \leftarrow ?$ for all $1 \leq j \leq n$.
2. Computing all $e_t = e_{t-1} + |x_t|$ and assigning all the explicit symbols, $w[e_t] \leftarrow x_t$ when $|x_t| = 1$, for all $1 \leq t \leq b$.
3. For each remaining unknown cell $w[j] = ?$ on a left-to-right pass over w, find the smallest k such that $w[\phi^k(j)] \neq ?$ in time $\mathcal{O}(k)$ and then fill $w[\phi^r(j)] \leftarrow w[\phi^k(j)]$ for all $0 \leq r < k$.

We now define and study macro-systems.

Definition 6.4.2 A *macro-system* is a tuple $M = (V, \Sigma, R, S)$, where V is a finite set of symbols called the *variables*, Σ is a finite set of symbols disjoint from V called the *terminals*, R is the set of rules (exactly one per variable)

$$R : V \to (V \cup \Sigma \cup \{A[i : j] \mid A \in V, i, j \in \mathbb{N}\})^*,$$

and $S \in V$ is the *initial variable*. If $R(A) = \alpha$ is the rule for A, we also write $A \to \alpha$. The symbols $A[i : j]$ are called *extraction symbols*. The rule $A \to \varepsilon$ is permitted only for $A = S$. The *size* of a macro-system is the sum of the lengths of the right-hand sides of the rules, $\mathtt{size}(M) = \sum_{A \in V} |R(A)|$.

We now define the string generated by a macro-system as the expansion of its initial symbol, $\mathtt{exp}(S)$. Such expansions are defined as follows.

Definition 6.4.3 Let $M = (V, \Sigma, R, S)$ be a macro-system. The *expansion* of a symbol is a string over Σ^* defined inductively as follows:

- If $a \in \Sigma$ then $\mathtt{exp}(a) = a$.
- If $S \to \varepsilon$, then $\mathtt{exp}(S) = \varepsilon$.
- If $A \to B_1 \cdots B_k$ is a rule, then $\mathtt{exp}(A) = \mathtt{exp}(B_1) \cdots \mathtt{exp}(B_k)$.
- $\mathtt{exp}(A[i : j]) = \mathtt{exp}(A)[i \mathinner{..} j]$.

We say that the macro-system is *valid* if there is exactly one solution $w \in \Sigma^*$ for $\mathtt{exp}(S)$. We only admit valid macro-systems, and say they *generate* w.

There are several reasons why a macro-system can be invalid. For example, the equations for $\mathtt{exp}(S)$ may have infinite solutions, as in $S \rightarrow S$ or $S \rightarrow S[1:2]$. It might also have no solutions, as in $S \rightarrow \mathtt{a}\,S$ or $S \rightarrow S[2:3]$. On the other hand, there can be valid solutions involving overlaps, like $S \rightarrow \mathtt{a}\,S[1:3]$, which solves to (only) $\mathtt{exp}(S) = \mathtt{aaaa}$.

Note that a macro-system looks very similar to a composition-system. The difference is that the latter impose an order to the variables so that each rule references only previous variables. Further, a run-length rule $A \rightarrow B^t$ can be translated in macro-systems as $A \rightarrow B\,A[1:(t-1)\cdot|\mathtt{exp}(B)|]$, therefore macro-systems are at least as powerful as collage-systems. The following example shows that they can be asymptotically strictly smaller.

Example 6.4.4 The smallest collage-system generating the Fibonacci string F_k (where $F_1 = b$, $F_2 = a$, and $F_{k+2} = F_{k+1}F_k$) is of size $\Theta(\log|F_k|)$ [102, Thm. 32]. Instead, we can mimic a BMS of size 4 [102, Lem. 35] with a constant-sized macro-system generating F_k, as follows (with $f_k = |F_k|$):

$$\begin{aligned} S &\rightarrow S[f_{k-2}+1, f_k-2]\ b\ a\ S[f_{k-2}+1, 2f_{k-2}] \text{ if } k \text{ is odd,} \\ S &\rightarrow S[f_{k-2}+1, f_k-2]\ a\ b\ S[f_{k-2}+1, 2f_{k-2}] \text{ if } k \text{ is even.} \end{aligned}$$

We now show how to decompress a macro-system. We note that, because there is no clear decompression order among the variables, expansion rules must be applied carefully for decompression, so that we expand only what is needed from the referenced variables.

Theorem 6.4.5 A macro-system $M = (V, \Sigma, R, S)$ can be decompressed, or determined to be invalid, in $\mathcal{O}(N)$ time and space, where $N = \sum_{A \in V} |\mathtt{exp}(A)|$.

Proof. We first determine the expansion lengths of all the variables, using the recurrence:

- $|\mathtt{exp}(a)| = 1$ if $a \in \Sigma$.
- $|\mathtt{exp}(A)| = |\mathtt{exp}(B_1)| + \cdots + |\mathtt{exp}(B_k)|$ if $A \rightarrow B_1 \cdots B_k$.
- $|\mathtt{exp}(A[i:j])| = j - i + 1$.

The expansion lengths are computed in time $\mathcal{O}(\mathtt{size}(M))$ by a simple procedure that recurses on the case $A \rightarrow B_1 \cdots B_k$. If this procedure falls in a loop, then the system is invalid. The reason is that we do not recurse on the extractions $\mathtt{exp}(A[i:j])$. Therefore, if we arrive again to $\mathtt{exp}(A)$ along the recursive expansion to compute $|\mathtt{exp}(A)|$, then $A \rightarrow^k X \cdot A \cdot Y$ for some $k > 0$, so either $|\mathtt{exp}(XY)| > 0$ and then the expansion of A is infinite, or $|\mathtt{exp}(XY)| = 0$ and $\mathtt{exp}(A)$ can be any string. In either case, M is invalid (note that an invalid system like $S \rightarrow S[2:3]$ will still pass this test, however).

Once the lengths are calculated, we create strings $E_A[1\,..\,|\mathtt{exp}(A)|]$ for all $A \in V$, with all their cells marked as unknown, $E_A[r] \leftarrow ?$ for all r. The decompression process will fill all

Algorithm 6 Defining a symbol when decompressing a macro-system.

```
Input  : Terminal or variable $A$ and position $r$ to define from $\texttt{exp}(A)$.
Output: Assigns $\texttt{exp}(A)[r]$ to $E_A[r]$ and any other position discovered along the process.
 1: function DEFINE$(A, r)$
 2:     if $r \notin [1\,..\,|\texttt{exp}(A)|]$ then return “invalid system” (out of bounds)
 3:     if $E_A[r] = \perp$ then return “invalid system” (loop detected)
 4:     if $E_A[r] \neq ?$ then return $E_A[r]$ (already known)
 5:     if $A = a$ (a terminal) then return $a$
 6:     $E_A[r] \leftarrow \perp$ (will be defined in the process)
 7:     if $A \rightarrow B_1 \cdots B_k$ (a variable) then
 8:         let $p$ be such that $\sum_{j=1}^{p-1} |\texttt{exp}(B_j)| < r \leq \sum_{j=1}^{p} |\texttt{exp}(B_j)|$
 9:         $E_A[r] \leftarrow \texttt{define}(B_p, r - \sum_{j=1}^{p-1} |\texttt{exp}(B_j)|)$
10:     else if $A = B[i : j]$ (an extraction) then
11:         $E_A[r] \leftarrow \texttt{define}(B, i + r - 1)$
12:     return $E_A[r]$
```

the necessary cells so that E_S has no unknown positions, at which point the decompressed string is $\texttt{exp}(S) = E_S$.

We successively define the symbols $E_S[1]$ to $E_S[|\texttt{exp}(S)|]$, which will trigger other definitions. The definition of a symbol $V[r]$ proceeds recursively, as shown in Algorithm 6. Note that we mark the traversed positions with $\perp$ to detect loops that flag the system as invalid. More importantly, although the recursion may visit many cells to define some $E_A[r] = c$, all those visited cells get assigned the value c as we return from the recursion. Since we define some new cell per unit of work, the total decompression cost is $\mathcal{O}(N)$, which absorbs $\mathcal{O}(\texttt{size}(M))$.

Line 8 of Algorithm 6 might suggest that we need a logarithmic-time binary search. We can, instead, precompute arrays $P_A[1\,..\,|\texttt{exp}(A)|]$ for every $A \rightarrow B_1 \cdots B_k$, so that we assign 1 to $P_A[1\,..\,|\texttt{exp}(B_1)|]$, 2 to $P_A[|\texttt{exp}(B_1)| + 1\,..\,|\texttt{exp}(B_1)| + |\texttt{exp}(B_2)|]$, and so on. After this $\mathcal{O}(N)$ space and time preprocessing, line 8 boils down to $p \leftarrow P_A[r]$. The sum of line 9 can be similarly precomputed in an array C_A of size $k \leq |\texttt{exp}(A)|$ for every A. □

Example 6.4.6 Now we show how to recover the string F_7 from the macro-system with rules

$$S \rightarrow S[6 : 11]ABS[6 : 10], A \rightarrow \texttt{a}, B \rightarrow \texttt{b}.$$

We first determine the expansion lengths of the variables of the macro-system: $|\texttt{exp}(A)| = 1$, $|\texttt{exp}(B)| = 1$, and $|\texttt{exp}(S)| = 13$. Then, we precompute the arrays

$$P_S[1\,..\,13] = [1, 1, 1, 1, 1, 1, 2, 3, 4, 4, 4, 4, 4] \text{ and } C_S[1\,..\,4] = [6, 7, 8, 13].$$

With these arrays precomputed, we initialize the array

$$E_S[1\,..\,13] = [?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?].$$

In a left-to-right fashion, we run Algorithm 6 to recover $E_S[i]$ for $i \in [1\,..\,13]$. We start by running $\texttt{define}(S, 1)$, which sets $E_S[1]$ to $\perp$, meaning that this cell is in the process of being defined (if possible). As S is a variable, we fall in line 7 of the algorithm. We obtain $p \leftarrow P_S[1] = 1$ and continue recursively with $\texttt{define}(S[6\,..\,11], 1)$. The algorithm then falls in line 10, and calls $\texttt{define}(S, 6)$. This call falls in line 7, hence we get $p = P_S[6] = 1$, and recursively call $\texttt{define}(S[6 : 11], 6)$. In this call, we fall again in line 10, which recursively calls $\texttt{define}(S, 11)$. The next recursive call is for $\texttt{define}(S, 8)$. At this point the array E_S is

$$E_S[1\,..\,13] = [\bot, ?, ?, ?, ?, \bot, ?, \bot, ?, ?, \bot, ?, ?].$$

The call for $\texttt{define}(S, 8)$ obtains $p = 3$ and calls $\texttt{define}(B, 1)$, which in turn calls $\texttt{define}(\texttt{b}, 1)$, which in line 5 returns b. This is assigned to $E_B[1]$ when we return to $\texttt{define}(B, 1)$, in line 9. As we return from the recursion to $\texttt{define}(S, 8)$, $\texttt{define}(S, 11)$, $\texttt{define}(S, 6)$, and finally $\texttt{define}(S, 1)$, the array E_S becomes

$$E_S[1\,..\,13] = [\texttt{b}, ?, ?, ?, ?, \texttt{b}, ?, \texttt{b}, ?, ?, \texttt{b}, ?, ?].$$

We now continue with $\texttt{define}(S, 2)$, which sets $E_S[1] = E_S[7] = E_A[1] = \texttt{a}$. The next call, for $\texttt{define}(S, 3)$, calls $\texttt{define}(S, 8)$, which this time returns b in line 4, as its value was already uncovered. The state of E_S is now

$$E_S[1\,..\,13] = [\texttt{b}, \texttt{a}, \texttt{b}, ?, ?, \texttt{b}, \texttt{a}, \texttt{b}, ?, ?, \texttt{b}, ?, ?].$$

We leave the completion of the other entries to the reader.

We now compare macro-systems, which can be decompressed in time $\mathcal{O}(N)$, with BMSs, which can be decompressed in time $\mathcal{O}(n)$. We define a restricted class of macro-systems we call *internal*, which turn out to be equivalent to BMSs, and can also be decompressed in time $\mathcal{O}(n)$.

Definition 6.4.7 A macro-system $M = (V, \Sigma, R, S)$ generating w is *internal* if every variable is reachable from S in the graph $G(V, E)$ where, if $A \rightarrow B_1 \cdots B_k \in R$, it holds that $(A, B_r) \in E$ for every variable (not terminal or extraction) B_r. We call $m(w)$ the size of the smallest internal macro-system representing w.

Intuitively, in an internal macro-system, the expansion $\texttt{exp}(A)$ of every variable A occurs in the string represented by the system. We first show the equivalence between internal macro-systems and BMSs; we show later how to decompress them in optimal time.

Theorem 6.4.8 Given a BMS of size b generating w, there exists an internal macro-system of size b generating w.

Proof. Let $(x_1, s_1), \ldots, (x_b, s_b)$ be the BMS generating $w = x_1 \cdots x_b$. We construct an internal macro-system $M = (\{S\}, \Sigma, R, S)$ with the single rule $S \rightarrow S_1 \cdots S_b$, where S_i is the single terminal x_i if $s_i = \bot$, and the extraction symbol $S[s_i, s_i + |x_i| - 1]$ if not. The system is valid because its only solution, for each j, is the explicit value of $w[j]$ if $\phi(j) = \bot$, or else $w[j] = w[\phi^{k-1}(j)]$ where $\phi^k(j) = \bot$ and thus $w[\phi^{k-1}(j)]$ is explicit in the macro-system. $\square$

Theorem 6.4.9 For every internal macro-system $M = (V, \Sigma, R, S)$ of size m generating w, there is a BMS of size at most m generating w.

Proof. We first compute $|\texttt{exp}(A)|$ for every variable $A \in R$ as done in the proof of Theorem 6.4.5. We then build a *pruned parse tree* for the macro-system, as follows. We start by creating the root and labeling it with the tuple $\langle S, 1, |\texttt{exp}(S)|\rangle$. The first time we create a node labeled $\langle A, l, r\rangle \in V$, with $A \to B_1 \cdots B_k \in R$, we create k children of the node, label them

$$\begin{aligned}&\langle B_1, l, l + |\texttt{exp}(B_1)| - 1\rangle,\\ &\langle B_2, l + |\texttt{exp}(B_1)|, l + |\texttt{exp}(B_1)| + |\texttt{exp}(B_2)| - 1\rangle, \ldots,\\ &\langle B_k, r - |\texttt{exp}(B_k)| + 1, r\rangle,\end{aligned}$$

and visit them recursively, left to right. In all other cases, that is, when A is a terminal symbol, an extraction symbol, or not the first occurrence of a variable, those nodes are leaves of the tree. If the macro-system is valid, this procedure will finish in time $\mathcal{O}(m)$ as in Theorem 6.4.5, and if the system is internal it will produce exactly one internal node per variable $A \in V$. It is then easy to see that the pruned parse tree has $m + 1$ nodes, $|V|$ of which are internal.

The procedure maintains the invariant that, if $\langle A, l, r\rangle$ labels a node of the pruned parse tree, then $w[l \mathinner{.\,.} r] = \texttt{exp}(A)$; we say that $\mathit{pos}(A) = l$ and note that this is the leftmost substring of w derived from A. The leaves of the pruned parse tree, $\langle X_1, l_1, r_1\rangle, \ldots, \langle X_b, l_b, r_b\rangle$ read from left to right, define a parse of w (note that X_t can be a variable, a terminal, or an extraction symbol).

Finally, we build a BMS for w, with one phrase per leaf label $\langle X_t, l_t, r_t\rangle$. If $X_t = a$ is a terminal, then the phrase is $(a, \perp)$, recording the explicit symbol. If $X_t = A$ is a variable, then the phrase is $(\texttt{exp}(A), \mathit{pos}(A))$, pointing to the leftmost substring derived from A. Finally, if $X_t = A[i : j]$ is an extraction symbol, then the phrase is $(\texttt{exp}(A)[i \mathinner{.\,.} j], \mathit{pos}(A) + i - 1)$, also pointing inside the leftmost substring of w derived from A (we detect that the macro-system is invalid if $j > |\texttt{exp}(A)|$).

The resulting BMS represents w and cannot have loops; otherwise there would be more no solution, or more than one solution, to the macro-system M and it would be invalid. The size of the BMS is $b \leq m$. □

That is, BMSs are equivalent to internal macro-systems. General macro-systems could be smaller in principle, though we have not found an example where this happens. (There exists an analogous situation with internal collage-systems [102].) It is now immediate that we can decompress internal macro-systems in linear time.

Corollary 6.4.10 Given an internal macro-system $M = (V, \Sigma, R, S)$ representing string $w[1 \mathinner{.\,.} n]$, we can compute w in time $\mathcal{O}(n)$.

Proof. The proof of Theorem 6.4.9 shows that we can produce, in time $\mathcal{O}(\texttt{size}(M))$, the parse $\langle X_1, l_1, r_1\rangle, \ldots \langle X_b, l_b, r_b\rangle$. From it, we build a macro-system $M' = (\{S\}, \Sigma, R', S)$ equivalent to M, that is, generating w, with the single rule $R' = \{S \to S_1 \cdots S_b\}$. The symbols S_t are only terminals and extractions: $S_t = a$ if $X_t = a$ is a terminal, $S_t = S[\mathit{pos}(A) : \mathit{pos}(A) + |\texttt{exp}(A)| - 1]$ if $X_t = A$ is a variable, and $S_t = S[\mathit{pos}(A) + i - 1 : \mathit{pos}(A) + j - 1]$

if $X_t = A[i:j]$ is an extraction symbol. We can now apply the decompression technique of Theorem 6.4.5 on M', which takes time $\mathcal{O}(N)$, as in M' it holds that $N = n$. In an internal macro-system it also holds that $\texttt{size}(M) \leq n$ because $|exp(A)| \geq k$ if $A \rightarrow B_1 \cdots B_k$. The total time is then $\mathcal{O}(n)$. □

6.5 NU-systems

A *NU-system*[1] is a formalism that generates a unique string in a way similar to an L-system, in the sense that terminals are not distinguished from variables and termination is defined by levels. The key difference is that, on the right-hand side of rules, a NU-system can have special extraction symbols of the form $A(l)[i:j]$, similar to the extractions symbols in macro-systems, whose meaning is to generate the l-th level from A, and then extract the substring starting at position i and ending at position j.

Definition 6.5.1 A *NU-system* is a tuple $U = (V, R, \Gamma, S)$, where V is a set of variables, $S \in V$ is the initial symbol, $\Gamma : V \rightarrow V$ is a coding, and R is a set of rules where the right-hand sides may contain *extractions*, that is, $R : V \rightarrow (V \cup E)^+$ with

$$E = \{A(l)[i:j] \mid A \in V \wedge l \in \mathbb{N} \wedge (i, j \in \mathbb{N} \vee -i, -j \in \mathbb{N})\}.$$

The symbol $A(l)[i:j]$ is *materialized* by expanding symbol A for l levels to obtain A^l, and then replacing $A(l)[i:j]$ by the substring $A^l[i\,..\,j]$ if i and j are positive, or by $A^l[|A^l| - i + 1\,..\,|A^l| - j + 1]$ if they are negative. Note that this may imply recursively materializing other extractions. We use $A(l)[:j]$ as a shorthand for $A(l)[1:j]$ if j is positive, and $A(l)[i:]$ as a shorthand for $A(l)[i:-1]$ if i is negative. The string represented by U is then $\Gamma(S^1)$.

Just as macro-systems, we will only consider *valid* NU-systems whose circular references can be solved by our decompression algorithm. This implies, by definition, that ν is reachable.

6.5.1 Decompression algorithm

The decompression process is akin to that of macro-systems, except that now we have several levels l for the same symbol A.

For every extraction $A(l)[i:j]$ in R we will prepare the strings $A^0, A^1, \ldots, A^l$, where $A^0 = A$, and their reverses $A^0_{rev}, A^1_{rev}, \ldots, A^l_{rev}$. Our goal is to determine S^1. We can determine the lengths of all the first levels, $|A^1| = |A^1_{rev}|$: Let $A \rightarrow B_1 \cdots B_k$, then it holds $|A^1| = \sum_{r=1}^{k} |B_r|$, where $|B_r| = 1$ if $B_r \in V$ and $|B_r| = j - i + 1$ if $B_r = B(l')[i:j] \in E$. The lengths of the following levels cannot be determined yet, as they depend on how the extractions will expand (we might never need to determine some of them along the decompression process).

We now define every A^1 and A^1_{rev} as follows. We start with an empty string A^1 and consider B_1 to B_r. If $B_r \in V$, we append B_r to A^1. If, instead, $B_r = B(l')[i:j]$, we append

[1] This is an enhanced version of the NU-systems defined in the conference version of this work [105, 106], and should be taken as the definitive one.

$B^{l'}[i] \cdot B^{l'}[i+1] \cdots B^{l'}[j]$ if i and j are positive, and $B^{l'}_{rev}[-j] \cdot B^{l'}_{rev}[-j+1] \cdots B^{l'}_{rev}[-i]$ if they are negative. We call these symbols *references*. We define A^1_{rev} analogously, putting the symbols and references in reverse order. If $l' = 1$ in a reference, it might be that some $B^1[k]$ or $B^1_{rev}[k]$ is already defined, in which case we replace the reference by its value. For every remaining reference $A^1[t]/A^1_{rev}[t] = B^{l'}[k]/B^{l'}_{rev}$, we set a *pointer* from the cell $B^{l'}[k]/B^{l'}_{rev}$ to $A^1[t]/A^1_{rev}[t]$. This pointer will be used later to copy the value of $B^{l'}[k]/B^{l'}_{rev}$ onto $A^1[t]/A^1_{rev}[t]$ when the former becomes known.

Once the strings A^1 and A^1_{rev} are defined in this way for all $A \in V$, we start defining the strings A^2 and A^2_{rev}. From left to right, for every $A^1[t] = B \in V$, we append B^1 to A^2. Note that B^1 includes symbols and references; both are appended to A^2 and the corresponding pointers to cells of A^2 are added (there may be several pointers leaving from a single cell). The process of scanning A^1 to form A^2 finishes when we hit some $A^1[t]$ that is a reference, because we do not yet know how it expands. Analogously, we define the maximal possible prefix of A^2_{rev} by scanning A^1_{rev} left to right. From the parts of A^2 and A^2_{rev} we could define, we also expand a maximal prefix of A^3 and A^3_{rev}, and so on until defining as much as possible from A^l and A^l_{rev}.

In the process, every time we define any symbol $A^k[t] \in V$ or $A^k_{rev}[t] \in V$, we check the possible pointers leaving that cell, and propagate the symbol to those cells. Those defined cells can trigger, recursively, further propagations by pointers, and also further expansions of prefixes, where we had stopped expanding because we had hit a reference that now has became a regular symbol.

We continue this process until either we completely define S^1 without references, or we have no further expansions to make and have not fully defined S^1. In the latter case, the NU-system is invalid. Because we define some cell of some A^k for each unit of work performed, we have the following result.

Theorem 6.5.2 A NU-system $U = (V, R, \Gamma, S)$ can be decompressed, or determined to be invalid, in time $\mathcal{O}(N)$, where $N = \sum_{A \in V} \sum_{k=0}^{l_A} |A^k|$, A^k is the expansion of A after k levels, and l_A is the maximum l value for an extraction $A(l)[i:j]$ found in R (with $l_A = 0$ if no extraction for A exists).

A simplified bound for the extraction time is given by $N = l_{\max} \sum_{A \in V} |A^{l_{\max}}|$, where $l_{\max} = \max_{A \in V} l_A$. Compared to the time to decompress a macro-system (Theorem 6.4.5), the time is now multiplied by the number of levels used.

Example 6.5.3 Consider the NU-system with rules

$$\begin{array}{lcl} A & \rightarrow & A\ B \\ B & \rightarrow & B \\ S & \rightarrow & A\ T(2)[2:4]\ S(3)[1:3]\ T \\ T & \rightarrow & S(1)[5:7]\ T(3)[1:3] \end{array}$$

We will omit the reversed symbols because there are no negative offsets. We first generate the level 1 as follows:

$$\begin{aligned}
A^1 &= A\ B \\
B^1 &= B \\
S^1 &= A\ T^2[2]\ T^2[3]\ T^2[4]\ S^3[1]\ S^3[2]\ S^3[3]\ T \\
T^1 &= S^1[5]\ S^1[6]\ S^1[7]\ T^3[1]\ T^3[2]\ T^3[3]
\end{aligned}$$

We now expand as much as possible the next levels, as follows (we omit A and B, which are trivial as they do not participate in extractions):

$$\begin{aligned}
S^2 &= A\ B\ \cdots \\
S^3 &= A\ B\ B \cdots
\end{aligned}$$

Since there are references to those new symbols, we can further complete S^1:

$$S^1 \ = \ A\ T^2[2]\ T^2[3]\ T^2[4]\ A\ B\ B\ T$$

And those newly defined symbols are referenced from T^1, which now becomes:

$$T^1 \ = \ A\ B\ B\ T^3[1]\ T^3[2]\ T^3[3]$$

This enables defining prefixes of T^2 and T^3:

$$\begin{aligned}
T^2 &= A\ B\ B\ B\ \cdots \\
T^3 &= A\ B\ B\ B\ B\ \cdots
\end{aligned}$$

With those, we can now complete S^1:

$$S^1 \ = \ A\ B\ B\ B\ A\ B\ B\ T$$

The string represented by the NU-system is then $\Gamma(ABBBABBT)$. Note that we could have decompressed the represented string even if there was a circular reference that did not affect S^1; for example if the rule for T was $T \rightarrow S(1)[5:7]\ T(3)[1:3]\ T(1)[7:8]$.

6.5.2 The measure ν

The smallest NU-system generating a string will define a new measure of repetitiveness we call ν.

Definition 6.5.4 The *size* of the NU-system $U = (V, R, \Gamma, S)$ is $\texttt{size}(U) = |V| + 1 + \sum_{A\in V} |R(A)|$, where the size of an expansion is taken as 4 when computing $|R(A)|$. We call $\nu = \nu(w)$ the size of the smallest NU-system generating w.

A first result stems from the fact that NU-systems encompass macro-systems and L-systems.

Theorem 6.5.5 It always holds that $\nu = \mathcal{O}(\min(\ell, b))$.

Proof. Given an L-system $L = (\Sigma, \varphi, \tau, s, d, n)$, we can define a NU-system $U = (V, R, \Gamma, S)$ as follows. Let $V = \Sigma \cup \{S\}$, where $S \notin \Sigma$, and $\Gamma = \tau$, and let $R = \varphi \cup \{S \to s(d)[1 : n]\}$. The NU-system will then expand d levels of s and extract the first n symbols to form S^1, and finally will apply $\Gamma = \tau$ to $S^1 = \varphi^d(s)[1\,..\,n]$. It is clear that U is valid, as it does not contain circular references. It then holds that $\nu = \mathcal{O}(\ell)$.

Consider now an internal macro-system $M = (V, \Sigma, R, S)$. By the proof of Corollary 6.4.10, we can convert M into a system with a single rule $S \to S_1 \cdots S_b$, where S_r is either a symbol of V or an extraction $S[i : j]$. We then construct a NU-system $U = (\{S'\} \cup \Sigma, R', \Gamma, S')$ where $\Gamma = \texttt{id}$ and $R' = \{S' \to S'_1 \cdots S'_b\}$: if $S_r = a \in \Sigma$, then $S'_r = a$; if instead $S_r = S[i : j]$, then $S'_r = S'(1)[i : j]$. It is clear that U generates the same string as M, and it is valid iff the macro-system M is valid. It then holds that $\nu = \mathcal{O}(b) = \mathcal{O}(m)$. □

An immediate corollary is that ν is uncomparable with δ.

Corollary 6.5.6 The measures δ and ν are uncomparable.

Proof. It follows because $\nu = \mathcal{O}(\ell)$ and $\ell = o(\delta)$ on some string families (Lemma 6.2.3), while on the other hand δ is unreachable on some string families and ν is always reachable. □

Finally, we show that NU-systems exploit the features of L-systems and macro-systems in a way that, for some string families, can reach sizes that are unreachable for both L-systems and BMSs independently.

Theorem 6.5.7 There exists a family of strings where $\nu = o(\min(\ell, b))$.

Proof. Let $\mathcal{K}_m$ be the family of strings of length m defined by Kociumaka et al. [79], needing $\Omega(\log^2 m)$ bits to be represented with any method (Def. 6.3.1), now over the alphabet $\{\texttt{0}, \texttt{1}\}$. We construct a new family $\mathcal{F} = \{x \cdot \mathbf{y}[1\,..\,m] \mid x \in \mathcal{K}_m\}$, where $\mathbf{y}$ is the infinite fixed point generated by the L-system utilized in Lemma 6.2.3. Hence, the strings in $\mathcal{F}$ have length $n = 2m$, and belong to $\{\texttt{0}, \texttt{1}, \texttt{a}, \texttt{b}, \texttt{c}\}^+$.

As shown in Lemma 6.3.2, it holds that $\ell = \Omega(\log^2 n / \log\log n)$ in $\mathcal{K}_m$. The same bound then holds on $\mathcal{F}$: if there is an L-system that generates an element in $\mathcal{F}$, we generate the corresponding prefix of $\mathcal{K}_m$ by changing the L-system prefix length from n to m. On the other hand, $b = \Omega(\sqrt{n})$ on $\mathcal{F}$, because $\delta = \Omega(\sqrt{n})$ on prefixes of $\mathbf{y}$ by Lemma 6.2.3,and δ is monotone with respect to the appending of prefixes or suffixes.

We now build a smaller NU-system for $\mathcal{F}$. Let x be a string in $\mathcal{K}_m$ with k symbols $\texttt{1}$. Let i_j be the number of $\texttt{0}$'s in x between the $(j-1)$-th and the j-th $\texttt{1}$'s, for $j \in [2\,..\,k]$. Also, let i_1 and i_{k+1} be the number of $\texttt{0}$'s at the left and right extremes of x. We construct the

NU-system $U = (V, R, \Gamma, S)$ as follows:

$$\begin{aligned}
V &= \{\mathtt{0}, \mathtt{1}, \mathtt{a}, \mathtt{b}, \mathtt{c}, S\} \\
R &= \{\mathtt{0} \rightarrow \mathtt{00}, \mathtt{1} \rightarrow \mathtt{1}, \mathtt{a} \rightarrow \mathtt{a}, \mathtt{b} \rightarrow \mathtt{ab}, \mathtt{c} \rightarrow \mathtt{cb}\} \\
&\quad \cup \{S \rightarrow \mathtt{0}(m)[: i_1]\mathtt{10}(m)[: i_2]\mathtt{1} \ldots \mathtt{0}(m)[: i_k]\mathtt{10}(m)[: i_{k+1}]\mathtt{c}(m)[: m]\} \\
\Gamma &= \{\mathtt{0} \rightarrow \mathtt{0}, \mathtt{1} \rightarrow \mathtt{1}, \mathtt{a} \rightarrow \mathtt{a}, \mathtt{b} \rightarrow \mathtt{b}, \mathtt{c} \rightarrow \mathtt{c}\}
\end{aligned}$$

By construction, this NU-system generates the string $x \cdot \mathbf{y}[: m]$ of length n, and its axiom has size $4(k+2)+k$, where $k = \Theta(\log n)$. Hence, it holds that ν is $\mathcal{O}(\log n)$ for these strings. Thus, $\nu = o(\min(\ell, b))$ in the family $\mathcal{F}$ we have constructed. □

NU-systems can then be smaller representations than those produced by any other compression method exploiting repetitiveness. This shows that combining copy-paste mechanisms with iterated morphisms is able, at least in principle, to further improve compression. On the other hand, finding the smallest NU-system is very likely NP-hard, and its extraction time is not bounded in terms of the size of the string that is generated.

6.5.3 Properties

We now study sensitivity, monotonicity, and other properties of NU-systems. We start showing that NU-systems grow nicely upon concatenations.

Proposition 6.5.8 If $w_1, w_2 \in \Sigma^*$, then $\nu(w_1 \cdot w_2) = \mathcal{O}(\nu(w_1) + \nu(w_2))$.

Proof. Let $U_1 = (\Sigma_1, R_1, \Gamma_1, S_1)$ and $U_2 = (\Sigma_2, R_2, \Gamma_2, S_2)$ be (minimal) NU-systems generating w_1 and w_2, respectively. Note that Σ_1 might be not disjoint from Σ_2. Then a NU-system $U = (\Sigma, R, \Gamma, S)$ generating $w_1 \cdot w_2$ can be built as follows. First, let $\Sigma'_k = \{\langle k, a\rangle \mid a \in \Sigma_k\}$, for $k = 1, 2$, be *marked* versions of the alphabets Σ_1 and Σ_2, so that $\Sigma'_1 \cap \Sigma'_2 = \emptyset$. The alphabet of U is then $\Sigma = \Sigma_1 \cup \Sigma_2 \cup \Sigma'_1 \cup \Sigma'_2 \cup \{S\}$, where S is a new initial symbol. For $k = 1, 2$, let R'_k be identical to R_k, except that each occurrence of $a \in \Sigma_k$ is replaced by $\langle k, a\rangle \in \Sigma'_k$. The rules of U are then the set

$$R = R'_1 \cup R'_2 \cup \{S \rightarrow \langle 1, S_1\rangle(1)[1 : |\langle 1, S_1\rangle^1|]\, \langle 2, S_2\rangle(1)[1 : |\langle 2, S_2\rangle^1|]\}.$$

Note that the lengths $|\langle 1, S_1\rangle^1| = |S_1^1|$ and $|\langle 2, S_2\rangle^1| = |S_2^1|$ are known because U_1 and U_2 are valid NU-systems. Finally, for $k = 1, 2$, let $\Gamma'_k : \Sigma'_k \rightarrow \Sigma_k$, so that $\Gamma'_k(\langle k, a\rangle) = \Gamma_k(a)$, and then $\Gamma = \Gamma'_1 \cup \Gamma'_2 \cup \{a \rightarrow a \mid a \in \Sigma_1 \cup \Sigma_2 \cup \{S\}\}$. It is easy to see that U generates $w_1 \cdot w_2$, and then $\nu(w_1 \cdot w_2) \leq \mathtt{size}(U) = \mathcal{O}(\mathtt{size}(U_1) + \mathtt{size}(U_2)) = \mathcal{O}(\nu(w_1) + \nu(w_2))$. □

This proposition shows, in particular, that NU-systems behave well upon appending and prepending of symbols.

Corollary 6.5.9 If $a \in \Sigma$ and $w \in \Sigma^*$, then $\nu(aw) \leq \nu(w) + \mathcal{O}(1)$ and $\nu(wa) \leq \nu(w) + \mathcal{O}(1)$.

Proof. It is (almost) a particular case of Proposition 6.5.8, where either $\nu(w_1) = \mathcal{O}(1)$ or $\nu(w_2) = \mathcal{O}(1)$ because $w_1 = a$ or $w_2 = a$. Instead of creating full new alphabets Σ'_1 and Σ'_2, we retain the alphabet of w and only create a special symbol for a (say, $\langle a \rangle$) and the rule $\langle a \rangle \to \langle a \rangle$, so that it is not modified along the derivation of w. The coding is then extended with the rule $\langle a \rangle \to a$. □

We now show that NU-systems are essentially monotonic, that is, one cannot obtain a smaller NU-system representing an extension of a string, except for constant additive factors.

Proposition 6.5.10 If $w \in \Sigma^*$ and $1 \leq i \leq j \leq |w|$, then $\nu(w[i\,..\,j]) \leq \nu(w) + \mathcal{O}(1)$.

Proof. Given $U = (\Sigma, R, \Gamma, S)$ generating w, the system $U' = (\Sigma', R', \Gamma', S')$ generates $w[i\,..\,j]$, where $S' \notin \Sigma$ is a new initial symbol, $\Sigma' = \Sigma \cup \{S'\}$, $R' = R \cup \{S' \to S(1)[i : j]\}$, and $\Gamma' = \Gamma \cup \{S' \to S'\}$. □

Those results imply that NU-systems behave well under edits on the represented string.

Corollary 6.5.11 If $w \in \Sigma^*$ and w' is obtained from w by applying one edit operation (insertion, deletion, or substitution of a symbol), then $\nu(w') \leq \nu(w) + \mathcal{O}(1)$.

Proof. All the edits on $w[1\,..\,n]$ can be expressed in terms of concatenating symbols or substrings of w: deleting the position i yields $w[1\,..\,i-1] \cdot w[i+1\,..\,n]$, substituting it by a yields $w[1\,..\,i-1] \cdot a \cdot w[i+1\,..\,n]$, and inserting a at position i yields $w[1\,..\,i-1] \cdot a \cdot w[i\,..\,n]$. Corollary 6.5.9 and Proposition 6.5.10 show how to build NU-systems of size $\nu(w) + \mathcal{O}(1)$ for all those expressions. For example, to insert a at position i we create rule $S' \to S(1)[1 : i-1]\,\langle a \rangle\, S(1)[i : n]$ for a new initial symbol S' and treat $\langle a \rangle$ as in Corollary 6.5.9. □

Proposition 6.5.12 If $w[1\,..\,n] \in \Sigma^*$ and $w' = w[n] \cdots w[1]$ is its reversal, then $\nu(w') = \nu(w)$.

Proof. We reverse all the rules, as well as the extractions $A(l)[i : j]$, which are reversed as $A'(l)[-j : -i]$, where A' denotes the reverse of A. By the symmetry of the decompression process, it is clear that the reversed system is valid as long as the original one is. □

6.6 Variants of L-systems and the Measure ℓ

In this section we study which features of L-systems are key for their compression power, and which are superfluous. We define and compare several classes of restricted L-systems and their corresponding compressibility measures. It turns out that all the natural restrictions to the L-system we consider yield reduced compression power.

First, we define the restricted classes of L-systems under scope.

Algorithm 7 Decompressing in real time the prolongable L-system $L = (\Sigma, \varphi, \tau, s, d, n)$; invoke with $\texttt{decompress}(s, n)$.

Input : Axiom s to expand, length to output $n > 0$.
Output: The string $\tau(\varphi^d(s))[1\,..\,n]$ with d large enough.

```
function DECOMPRESS($s, n$)
    $w[1] \leftarrow s$
    $r \leftarrow 1$
    $p \leftarrow 1$
    while true do
        let $w[r] \rightarrow b_1 \cdots b_k \in \varphi$
        $r \leftarrow r + 1$
        for $i \leftarrow 1$ to $k$ do
            $w[p] \leftarrow b_i$
            output $\tau(b_i)$
            if $p = n$ then return
            $p \leftarrow p + 1$
```

Definition 6.6.1 Let $L = (\Sigma, \varphi, \tau, s, d, n)$ be an L-system. We say L is *expanding* when φ is expanding. We say L is *k-uniform* for some $k \geq 2$, or just *uniform*, when φ is k-uniform. We say L is *prolongable*, if φ is prolongable on s (i.e., if $s \rightarrow sx$ is a rule, with $x \neq \varepsilon$). We say L is *codingless* if $\tau = \texttt{id}$ (i.e., the identity coding).

We define some compressibility measures based on L-systems that satisfy some of those restrictions.

Definition 6.6.2 The measure $\ell_{\mathtt{e}}(w)$ (resp., $\ell_{\mathtt{u}}(w)$) denotes the size of the smallest expanding (resp., uniform) L-system generating w. The measure $\ell_{\mathtt{p}}(w)$ denotes the size of the smallest prolongable L-system generating w. The measure $\ell_{\mathtt{c}}(w)$ denotes the size of the smallest codingless L-system generating w. The measure $\ell_{\mathtt{pc}}(w)$ denotes the size of the smallest prolongable and codingless L-system generating w. The measure $\ell_{\mathtt{pu}}(w)$ denotes the size of the smallest prolongable and uniform L-system generating w.

It is known that different classes of L-systems produce different classes of languages and infinite words [111]. Some of these classes also differ in the factor complexity of the sequences they can generate [112]. It is interesting to understand how these differences in terms of expressive power and factor complexity translate into the compression power of the L-systems. In particular, prolongable L-systems generate a class of infinite words called morphic words; codingless and prolongable L-systems generate the class of purely morphic words; and prolongable uniform L-systems generate the so-called automatic words [4]. So even if restricting L-systems reduces their compression power, they still can be useful to compress prefixes of infinite words in these classes, and working on them may be more efficient than on general L-systems.

For example, in a prolongable system (i.e., using $\mathcal{O}(\ell_{\mathtt{p}})$ space), Algorithm 7 shows how to decompress the represented string in *real time* (i.e., each successive symbol of w is written in $\mathcal{O}(1)$ time). As another example, in an expanding system (i.e., using $\mathcal{O}(\ell_{\mathtt{e}})$ space) we can always limit the depth to $\lceil \log_2 n \rceil$, by starting from the axiom $f^{d-\lceil \log_2 n \rceil}(s)$, and obtain optimal $\mathcal{O}(n)$ decompression time using just Algorithm 2. The cost for extracting $w[i \mathinner{..} j]$ we obtained in Section 6.1.2 is also reduced, to $\mathcal{O}(|\varphi| \log n) = \mathcal{O}(\ell_e \log n)$ preprocessing space and time, and $\mathcal{O}(j - i + \log n)$ extraction time. Further, if a system is k-uniform (i.e., using $\mathcal{O}(\ell_{\mathtt{u}})$ space), then we know easily the size to which every symbol expands after l levels, in which case we can efficiently extract $w[i \mathinner{..} j]$ without need of any preprocessing or extra space: in lines 6–7 of Algorithm 5 we simply use $r \leftarrow \lceil i/k^l \rceil$ and $i \leftarrow 1 + ((i-1) \bmod k^l)$. The time is then $\mathcal{O}(j - i + d)$. Further, if $d > \lceil \log_k n \rceil$, we can slightly modify the L-system so that its axiom is $f^{d-\lceil \log_k n \rceil}(s)$, as explained, and obtain extraction time $\mathcal{O}(j - i + \log_k n)$.

As a consequence, we can upper bound the size g of the smallest grammar with respect to the measure $\ell_{\mathtt{e}}$ (and $\ell_{\mathtt{u}}$). To do so, we observe that we can always simulate an L-system L with depth d, with a CFG of size $\mathcal{O}(d \cdot \mathtt{size}(L))$. As the value d can be bounded for expanding and uniform L-systems, we obtain the following result.

Lemma 6.6.3 For any L-system L, there exists a CFG G of size $|G| = \mathcal{O}(d \cdot \mathtt{size}(L))$ generating the same string. Further, it always holds that $g = \mathcal{O}(\ell_{\mathtt{e}} \log n)$.

Proof. Let $L = (\Sigma, \varphi, \tau, s, d, n)$ be an L-system generating $w[1 \mathinner{..} n]$. Consider the derivation tree of L, which is obtained as follows: the root is a node labeled s at depth 0. If A is a node at depth $i \in [0 \mathinner{..} d-2]$, then the children of A at depth $i+1$ are the symbols in $\varphi(A)$ read from left to right. For $i = d-1$, the children of A are the symbols in $\tau(\varphi(A))$ read from left to right. The nodes at each depth i spell out a string L_i, where $L_0 = s$ and $L_d = w$. We create a CFG $G = (V, \Sigma, R, S)$ that simulates L as follows. The set V contains, for each variable $A \in \Sigma$ of the L-system, d nonterminals $A_0, \ldots, A_{d-1}$. The terminals of the grammar are the set of L-system variables, that is, Σ. Then, for each L-system rule $A \rightarrow B_1 \cdots B_k$ appearing at depth $0 \leq i \leq d-2$ of the L-system, we add $A_i \rightarrow (B_1)_{i+1} \cdots (B_k)_{i+1}$ to the set of rules R. Further, for each rule $A \rightarrow B_1 \cdots B_k$ appearing at depth $d-1$ in L, we add the grammar rule $A_{d-1} \rightarrow \tau(B_1) \cdots \tau(B_k)$ to R (this is well defined because each B_i belongs to Σ). Finally, the initial symbol of G is $S = s_0$. Note that the derivation trees of G and L are topologically identical and spell the same string at depth d. Hence, the grammar G is of size at most $(d+1) \cdot \mathtt{size}(L)$ and generates a string w^+, of which the desired string $w[1 \mathinner{..} n]$ is a prefix.

We now modify G to generate exactly $w[1 \mathinner{..} n]$. The idea is to create a new nonterminal per level L_i that will expand to a prefix of the string some nonterminal of that level expands to. Our new initial symbol (of level 0) will be $S' = s'_0$, whose expansion must be pruned to length $l_0 = n$. In general, given a nonterminal $A_i \rightarrow (B_1)_{i+1} \cdots (B_k)_{i+1}$ whose expansion must be pruned to length l_i, we define k_i as the maximum position $j < k$ such that $|\mathtt{exp}((B_1)_{i+1} \cdots (B_j)_{i+1})| < l_i$. We then need to fully expand the symbols $(B_1)_{i+1} \cdots (B_{k_i})_{i+1}$, and then expand a prefix of length $l_{i+1} = l_i - |\mathtt{exp}((B_1)_{i+1} \cdots (B_{k_i})_{i+1})|$ of $(B_{k_i+1})_{i+1}$. We therefore create a new rule $A'_i \rightarrow (B_1)_{i+1} \cdots (B_{k_i})_{i+1} \cdot (B_{k_i+1})'_{i+1}$, and recursively continue in level $i+1$ with the task of creating a variant $(B_{k_i+1})'_{i+1}$ of $(B_{k_i+1})_{i+1}$ whose expansion is pruned to length l_{i+1}. In the process we at most double the size of G, which is thus of size $\mathcal{O}(d \cdot \mathtt{size}(L))$.

For the second claim, if an L-system is expanding and $d > \log n$, then the prefix $w[1 \mathinner{..} n]$

of L_d is generated from the first symbol of $L_{d-\lceil \log n \rceil}$, which can then be made the axiom and d reduced to $\lceil \log n \rceil$. In this case, the grammar G produced is of size $\mathcal{O}(\texttt{size}(L) \log n)$. Thus, $g = \mathcal{O}(\ell_{\mathtt{e}} \log n)$. □

As $\ell_{\mathtt{c}} = \mathcal{O}(g)$ by Lemma 6.2.2 (the coding used in the proof has $\tau = \mathtt{id}$), and $g = \mathcal{O}(\ell_{\mathtt{e}} \log n)$ by Lemma 6.6.3, we obtain the following corollary.

Corollary 6.6.4 It always holds that $\ell_{\mathtt{c}} = \mathcal{O}(\ell_{\mathtt{e}} \log n)$.

Surprisingly, all the restricted L-systems (except possibly uniform systems) outperform δ on some string family. We already showed this for $\ell_{\mathtt{pc}}$ (and thus $\ell_{\mathtt{p}}$ and $\ell_{\mathtt{c}}$) in Lemma 6.2.3, where the L-system we used was prolongable and without coding. The next lemma proves that the same holds for $\ell_{\mathtt{e}}$.

Lemma 6.6.5 There exists a string family where $\ell_{\mathtt{e}} = \mathcal{O}(1)$ and $\delta = \Omega(\log n)$.

Proof. We use a small modification of the D0L-sequence described by Ehrenfeucht et al. [39, Lemma 5]. For simplicity, let d be a power of 16, and define the following expanding L-system:

$$\begin{aligned} L = (&\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, \\ &\{\mathtt{a} \to \mathtt{a}^2, \mathtt{b} \to \mathtt{b}^{16}, \mathtt{c} \to \mathtt{cbab}\}, \\ &\tau = \mathtt{id}, \\ &s = \mathtt{c}, \\ &d, \\ &n = |\varphi^{(\log_2 d)+1}(s)|). \end{aligned}$$

By definition $\ell_{\mathtt{e}} = \mathcal{O}(1)$ in this family of strings (where we vary d; by the formula of n, the distinct elements of the family are obtained for values of d that are powers of 16). Let $x = \mathtt{bab}$. The string generated by this system is $w = \mathtt{c}x\varphi(x)\varphi^2(x)\cdots\varphi^{\log_2 d}(x)$. Note that we use prefix truncation to obtain a string that is orders of magnitude shorter than $\varphi^d(s)$. We do this to ensure that the value d is large enough with respect to n. Now consider the images of the form $\varphi^i(x)$ for $i \geq 1$. First note that $\varphi^i(x) = \mathtt{b}^{16^i}\mathtt{a}^{2^i}\mathtt{b}^{16^i}$ and its length is $2 \cdot 16^i + 2^i$. The length of the string w is then $n = 1 + \sum_{i=1}^{\log_2 d}(2 \cdot 16^i + 2^i) = \Theta(d^4)$. Therefore, $d = \Theta(\sqrt[4]{n})$.

We now show that $\delta = \Omega(\log d)$. To do so, we get a lower bound on the number of length-d substrings of the form $\mathtt{b}^p\mathtt{a}^{2^u}\mathtt{b}^q$. The string $\varphi^i(x)$ is a substring of w, for $i \in [1\,..\log_2 d]$. In particular, a length-d factor of the required form can appear inside $\varphi^i(x)$ only if $|\varphi^i(x)| \geq d$. This condition is verified if $i \geq \log_{16} d$. Observe that $\varphi^{\log_{16} d}(x)$ contains $2^{\log_{16} d} = d^{\log_{16} 2} < d$ $\mathtt{a}$'s, that $\varphi^{\log_2 d}(x)$ contains d $\mathtt{a}$'s, and that both strings contain at least d $\mathtt{b}$'s at each side of the $\mathtt{a}$'s. Hence, for each $i \in [\log_{16} d\,..\log_2 d]$ we can slide a window of length d containing $\mathtt{a}^{2^i}$ starting at every possible position, surrounded by $\mathtt{b}$'s. This yields

$$\sum_{i=\log_{16} d}^{\log_2 d} (d - 2^i + 1) = \Theta(d \log d)$$

distinct substrings of length d. Thus, $\delta = \Omega(\log d) = \Omega(\log \sqrt[4]{n}) = \Omega(\log n)$. □

In the rest of the section we show that, despite still breaking the barrier of δ for some string families, each of the restrictions we can put to L-systems reduces their compression power, so all the features we have included in L-systems are needed to reach the measure ℓ.

We start by showing that ℓ can be asymptotically strictly smaller than $\ell_{\texttt{p}}$. That is, restricting L-systems to be prolongable yields a weaker measure. We will actually prove that $\ell_{\texttt{p}}$ can be asymptotically larger than $\ell_{\texttt{c}}$, the L-systems without codings.

Lemma 6.6.6 There exists a string family where $\ell_{\texttt{p}} = \Omega(\ell_{\texttt{c}} \log n / \log\log n)$.

Proof. Let $\mathcal{F} = \{\texttt{a}^{n-1}\texttt{b} \,|\, n \geq 1\}$. Clearly, $\ell_{\texttt{c}}$ is constant in this string family: the L-system $L_n = (\Sigma, \varphi, \tau, s, d, n)$ where $\Sigma = \{\texttt{a}, \texttt{b}\}, \varphi = \{\texttt{a} \rightarrow \texttt{a}, \texttt{b} \rightarrow \texttt{ab}\}, \tau = \texttt{id}, s = \texttt{b},$ and $d = n - 1$ produces each string in $\mathcal{F}$ by changing only the value of n accordingly. Note that these L-systems are not prolongable.

Now let $L_n = (\Sigma_n, \varphi_n, \tau_n, s, d_n, n)$ be the smallest prolongable L-system generating $\texttt{a}^{n-1}\texttt{b}$. Let $k = |\Sigma_n|$ and $t = \texttt{width}(\varphi_n) > 1$. Observe that it is only necessary to have one symbol $c \in \Sigma_n$ with $\tau_n(c) = \texttt{b}$ because there is only one $\texttt{b}$ in $\texttt{a}^{n-1}\texttt{b}$, so w.l.o.g. assume that $\texttt{b} \in \Sigma_n$ and $\tau_n(\texttt{b}) = \texttt{b}$. As the system is prolongable, each level is a prefix of the next one. This implies that the morphism should be iterated until $\texttt{b}$ appears for the first time, and then we can safely extract the prefix. This must happen in the first k iterations of the morphism; otherwise, $\texttt{b}$ is not reachable from s. The reason is that, if an iteration does not yield a new symbol, then no new symbols will appear since then, and there are no more than k symbols. Once $\texttt{b}$ appears, it cannot be deleted in the following levels, so it cannot appear before position n. Hence, $t^k \geq n$, implying $k \geq \log_t n$. By definition, $\ell_{\texttt{p}} \geq k \geq \log_t n$ and $\ell_{\texttt{p}} \geq t$, so $\ell_{\texttt{p}} \geq \max(t, \log_t n)$. This is $\Omega(\log n / \log\log n)$: if $t \leq \log n / \log\log n$, then $\log_t n \geq \log n / \log\log n$. Thus, $\ell_{\texttt{p}} = \Omega(\ell_{\texttt{c}} \log n / \log\log n)$ in this string family. □

We can prove a similar result for uniform systems.

Lemma 6.6.7 There exists a string family where $\ell_{\texttt{p}} = \Omega(\ell_{\texttt{u}} \log n / \log\log n)$.

Proof. It is not difficult to see that $\ell_{\texttt{u}}$ is constant in the family $\{\texttt{a}^{2^k}\texttt{b} \,|\, k \geq 0\}$: consider the axiom $s = \texttt{c}$ and the rules $\texttt{c} \rightarrow \texttt{ab}$, $\texttt{a} \rightarrow \texttt{aa}$, $\texttt{b} \rightarrow \texttt{bb}$, the level $d = k$ and the prefix length $n = 2^k + 1$. The same argument as in Lemma 6.6.6 yields that $\ell_{\texttt{p}} = \Omega(\ell_{\texttt{u}} \log n / \log\log n)$ for this string family. □

We now show that if we remove the coding from prolongable L-systems (which corresponds to the variant $\ell_{\texttt{pc}}$) we end with a much worse measure. We change the usual alphabet for clarity of presentation.

Lemma 6.6.8 There exists a string family where $\ell_{\texttt{pc}} = \Omega(\ell_{\texttt{p}}\sqrt{n})$.

Proof. We prove that $\ell_{\texttt{pc}} = \Theta(n)$ on $\mathcal{F} = \{\texttt{0}^{n-1}\texttt{1} \,|\, n \geq 2\}$, whereas $\ell_{\texttt{p}} = \mathcal{O}(\sqrt{n})$. Any minimal codingless prolongable L-system generating $\texttt{0}^{n-1}\texttt{1}$ must contain the rule $\texttt{0} \rightarrow \texttt{0}^{n-1}\texttt{1}$, which implies $\ell_{\texttt{pc}} = \Theta(n)$. This is because if the L-system is prolongable and the coding is the identity: i) the initial symbol must be $s = \mathtt{0}$ as it will appear as a prefix of all the following iterations; ii) in the prolongable rule $\mathtt{0} \to \mathtt{0}x$, if $|\varphi(\mathtt{0})| \leq n$, then the non-empty string x can contain only 0s and 1s, otherwise undesired symbols would appear in the final string. If x does not contain 1s, then 1 is unreachable from 0, which is a contradiction. So, it must be the case that x contains a least one 1, and the first of them has to be at position n.

On the other hand, we can construct a prolongable L-system for $\mathtt{0}^{n-1}\mathtt{1}$, with its coding defined as $\tau(\mathtt{1}) = \mathtt{1}$ and $\tau(c) = \mathtt{0}$ for every other symbol $c \neq \mathtt{1}$ as follows: Let $n - 1 = k\lfloor\sqrt{n-1}\rfloor + j$ with $\lfloor\sqrt{n-1}\rfloor > 3, k > 1$, and $0 \leq j < \lfloor\sqrt{n-1}\rfloor$ (k and j are integers). We assume n is sufficiently big so the constraints are satisfied. Then, we define the following rules

$$\begin{aligned}
\mathtt{a} &\to \mathtt{ab} \\
\mathtt{b} &\to \mathtt{c}^{k-1}\mathtt{d} \\
\mathtt{c} &\to \mathtt{0}^{\lfloor\sqrt{n-1}\rfloor-1} \\
\mathtt{d} &\to \mathtt{0}^{\lfloor\sqrt{n-1}\rfloor-3+j}\mathtt{1},
\end{aligned}$$

and set the initial symbol $s = \mathtt{a}$. The first four levels of the L-system before applying the coding τ are

$$\begin{aligned}
\varphi^0(\mathtt{a}) &= \mathtt{a} \\
\varphi^1(\mathtt{a}) &= \mathtt{ab} \\
\varphi^2(\mathtt{a}) &= \mathtt{abc}^{k-1}\mathtt{d} \\
\varphi^3(\mathtt{a}) &= \mathtt{abc}^{k-1}\mathtt{d}\mathtt{0}^{(\lfloor\sqrt{n-1}\rfloor-1)(k-1)}\mathtt{0}^{\lfloor\sqrt{n-1}\rfloor-3+j}\mathtt{1},
\end{aligned}$$

and it can be verified that

$$|\varphi^3(\mathtt{a})| = 3 + (k-1) + (\lfloor\sqrt{n-1}\rfloor - 1)(k-1) + (\lfloor\sqrt{n-1}\rfloor - 3 + j) + 1 = n.$$

Moreover, we can deduce from the observation above that $\tau(\varphi^3(\mathtt{a})) = \mathtt{0}^{n-1}\mathtt{1}$, as only the symbol 1 is mapped to 1 by the coding. The claimed L-system is then $L = (\{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{0},\mathtt{1}\}, \varphi, \tau, s = \mathtt{a}, 3, n\})$, and it generates $\mathtt{0}^{n-1}\mathtt{1}$ as required, for n bigger than some constant. The size of the L-system is clearly $\Theta(\sqrt{n})$. □

By using the same family above, the following corollary holds.

Corollary 6.6.9 There exists a string family where $\ell_{\mathtt{pc}} = \Omega(\ell_{\mathtt{c}} n)$.

Proof. Just note that $\ell_{\mathtt{c}}$ is constant in the family used in Lemma 6.6.8. □

It is surprising that this weak measure $\ell_{\mathtt{pc}}$ can be much smaller than δ for some string families, as mentioned before. On the other hand, it does not hold that $\ell_{\mathtt{pc}} = \mathcal{O}(g)$ for every string family, because $g = \Theta(\log n)$ on $\{\mathtt{0}^{n-1}\mathtt{1} \mid n \geq 1\}$.

Corollary 6.6.10 The measure $\ell_{\mathtt{pc}}$ is uncomparable to the measures δ and g.

If we restrict L-systems to be expanding, we also end up with a weaker measure. This shows that, in general, it is not possible to transform L-systems into expanding ones without incurring an increase in size.

Proposition 6.6.11 There exists a string family where $\ell_{\mathsf{e}} = \Omega(\ell_{\mathsf{pc}}\sqrt{n}/\log n)$.

Proof. Such a family is the one of Lemma 6.2.3. In this family, $\ell_{\mathsf{pc}} = \mathcal{O}(1)$. On the other hand, from Lemma 6.6.3, it holds that $\ell_{\mathsf{e}} = \Omega(g/\log n) \subseteq \Omega(\delta/\log n)$. Recall that $\delta = \Theta(\sqrt{n})$ in this string family, so $g = \Omega(\sqrt{n})$. (Further, a grammar of size $g = \mathcal{O}(\sqrt{n})$ is easily obtained by setting $A_0 \to \mathtt{b}$, $A_{i+1} \to \mathtt{a}\,A_i$, and the initial rule $S \to \mathtt{c}A_0A_1\cdots A_{d-1}$.) Hence, $\ell_{\mathsf{e}} = \Omega(\sqrt{n}/\log n) = \Omega(\ell_{\mathsf{pc}}\sqrt{n}/\log n)$. □

Finally, we show that L-systems can be asymptotically strictly smaller than codingless L-systems on some string families.

Lemma 6.6.12 There exists a string family where $\ell = o(\ell_{\mathsf{c}})$.

Proof. Let $\mathcal{F} = \{\mathtt{bba}^k\mathtt{ba}^{2^k} \mid k \geq 1\}$. The L-system $L_l = (\Sigma_l, \varphi_l, \tau, \mathtt{d}, k, 2^k + k + 2)$ where

$$\Sigma_l = \{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d}\} \qquad \varphi_l : \begin{cases} \mathtt{a} & \mapsto & \mathtt{a} \\ \mathtt{b} & \mapsto & \mathtt{b} \\ \mathtt{c} & \mapsto & \mathtt{cc} \\ \mathtt{d} & \mapsto & \mathtt{bbabcc} \end{cases} \qquad \tau : \begin{cases} \mathtt{a} & \mapsto & \mathtt{a} \\ \mathtt{b} & \mapsto & \mathtt{b} \\ \mathtt{c} & \mapsto & \mathtt{a} \\ \mathtt{d} & \mapsto & \mathtt{b} \end{cases}$$

can generate each word in $\mathcal{F}$ by changing k. Hence, $\ell = \mathcal{O}(1)$ in this family.

We now show that any codingless L-system generating $\mathtt{bba}^k\mathtt{ba}^{2^k}$ has size $\omega(1)$. Assume for the sake of contradiction that a constant-size codingless L-system $L = (\Sigma, \varphi, \mathtt{id}, s, d, n)$ generates $w = \mathtt{bba}^k\mathtt{ba}^{2k}$. Then, there exists a constant α such that $|\Sigma| \leq \alpha$ and $\mathtt{width}(\varphi) \leq \alpha$. The longest string L could generate would have length α^d. Hence, $d = \Omega(\log n) = \Omega(k)$.

Let $b_0, b_1, \ldots, b_d$ be the sequence of the first symbols of $\varphi^i(s)$, for $0 \leq i \leq d$ (so $b_0 = s$). By the pigeonhole principle, for sufficiently big values of k (and consequently big values of d), this sequence has a period of length q starting from b_p, with $p + q \leq \alpha \leq d$. Then there exist indexes t and j such that $t = d - jq$ and $p \leq t < p + q$. By the q-periodicity of the sequence starting at b_t, it holds that $\varphi^q(b_t) = b_t x$ for some (possibly empty) string x. Moreover, as there is no coding, it must be that $b_t = b_d = \mathtt{b}$.

Let us then define a new L-system $L' = (\Sigma \cup \{s'\}, \varphi', \mathtt{id}, s', d', n)$, with a new (initial) symbol $s' \to \varphi^t(s)$ and otherwise $\varphi' = \varphi^q$; moreover $d' = 1 + ((d - t)/q)$. Clearly, L' generates w and there is also a constant α' such that $|\Sigma| + 1 \leq \alpha'$ and $\mathtt{width}(\varphi') \leq \alpha'$. There are two possibilities: (i) $|\varphi'(\mathtt{b})| > 1$ or (ii) $|\varphi'(\mathtt{b})| = 1$. In case (i), we have $\varphi'(\mathtt{b}) = \mathtt{bb}x$ for a possibly empty string x of bounded length, because $(\varphi')^{d'}[2] = \mathtt{b} = \varphi'(\mathtt{b})[2]$. As there is no coding and φ' is prolongable on $\mathtt{b}$, the image $\varphi'(\mathtt{b})$ is a prefix of $\varphi'^{d'}(s')$, and so is $\varphi'^2(\mathtt{b}) = \mathtt{bb}x\mathtt{bb}x\varphi'(x)$. This is a contradiction for large enough k, as the third $\mathtt{b}$ appears after just a constant number of symbols. Therefore the only possible case is (ii), that is, $\varphi'(\mathtt{b}) = \mathtt{b}$. This implies that $\mathtt{b}$ is part of a cycle in the original morphism φ.

We now reason analogously on the *third* symbol of the derivation from s'. Let $b'_0, b'_1, \ldots, b'_{d'}$ be the sequence of the third symbols of $(\varphi')^i(s')$, for $0 \le i \le d'$. Then, since α' is a constant, there must exist a period of length q' starting at $b'_{p'}$, with $p' + q' \le \alpha' \le d'$, and the corresponding values $t' = d' - j'q'$ and $p' \le t' < p' + q'$, so that $(\varphi')^{q'}(b'_{t'}) = b'_{t'}x'$ for some possibly empty string x'. Because there is no coding, it must be that $b'_{t'} = \mathtt{a}$ and that x is a prefix of $\mathtt{a}^{k-1}\mathtt{ba}^{2^k}$. If x were non-empty, it would still have its length bounded by a constant. Hence, for sufficiently big values of k, it must be that $x = \mathtt{a}^r$ for some $r \ge 1$. Therefore, $\varphi'^{j'q'}(\mathtt{a})$ yields $\Omega(2^{d'})$ $\mathtt{a}$'s in the first run, which is a contradiction. Thus, $(\varphi')^{q'}(\mathtt{a}) = \mathtt{a}$. This also implies that $\mathtt{a}$ belongs to a cycle in the original morphism φ.

We shift our attention again to the morphism φ. We now prove that $d = \mathcal{O}(k)$. We note that $\varphi^d(s)$ must contain a run of exactly k $\mathtt{a}$'s. Since $|\varphi^j(\mathtt{a})| = 1$ for every j, there must be some other symbol $\mathtt{c}$ in the derivation of the run such that, for some constant t, $\varphi^t(\mathtt{c})$ contains at least one $\mathtt{a}$ and at least one $\mathtt{c}$; otherwise the constant-sized system cannot generate an arbitrary number of $\mathtt{a}$'s. But then, there are $\Omega(d/t) = \Omega(d)$ occurrences of $\mathtt{a}$ in the run; hence $d = \mathcal{O}(k)$.

In the following we use some definitions and known results by Salomaa [123], who studied the growth rates of D0L-systems. A letter c is said to be *expanding* on φ if there exists j such that $\varphi^j(c) = xcycz$. A codingless L-system has exponential growth with d if and only if an expanding letter appears in its derivation [123, Thm. 1]. For convenience, we extend this definition so that a letter c is also expanding when $\varphi^j(c) = xc'y$ for some j, and c' is expanding, that is, if we consider c as the axiom of the system, then it has exponential growth. This extension implies that any expanding symbol contains at least one expanding symbol in its image under φ. As $d = \Theta(k)$, the only way this system could possibly generate a string of length over 2^k, is that the system uses an expanding letter in its derivation. Hence, s has to be expanding using our extended definition. Note also that $\mathtt{a}$ and $\mathtt{b}$ are not expanding, as they belong to single-symbol cycles.

We construct two sequences, $c_0, c_1, \ldots, c_d$ and $x_1, \ldots, x_d$, such that $c_0 = s$, c_i is expanding, x_i does not contain expanding letters, and $\varphi(c_i) = x_{i+1}c_{i+1}y_{i+1}$ (i.e., c_{i+1} is the first expanding symbol in the image of c_i). It is clear that

$$\varphi^d(s) = \varphi^{d-1}(x_1)\varphi^{d-2}(x_2)\cdots\varphi(x_{d-1})\cdot(x_d \cdot c_d \cdot y_d)\cdot\varphi(y_{d-1})\cdots\varphi^{d-2}(y_2)\varphi^{d-1}(y_1).$$

Note that the strings x_i have length bounded by the constant $\mathtt{width}(\varphi)$ and no expanding symbols. Hence, φ grows polynomially on them. On the other hand, c_d is an expanding symbol, hence distinct from $\mathtt{a}$ and $\mathtt{b}$, appearing at a position $o(2^k)$. This yields a contradiction for sufficiently big values of k. □

We have shown that imposing restrictions on the length of the rules of an L-system, forcing them to be prolongable, or removing the coding, does impact their compression power. On the other hand, these restricted L-systems may simplify and speed up some relevant processes like decompressing or direct accessing the represented string. We summarize the results of this section in Figure 6.3, which also includes the measure ν from Section 6.4.

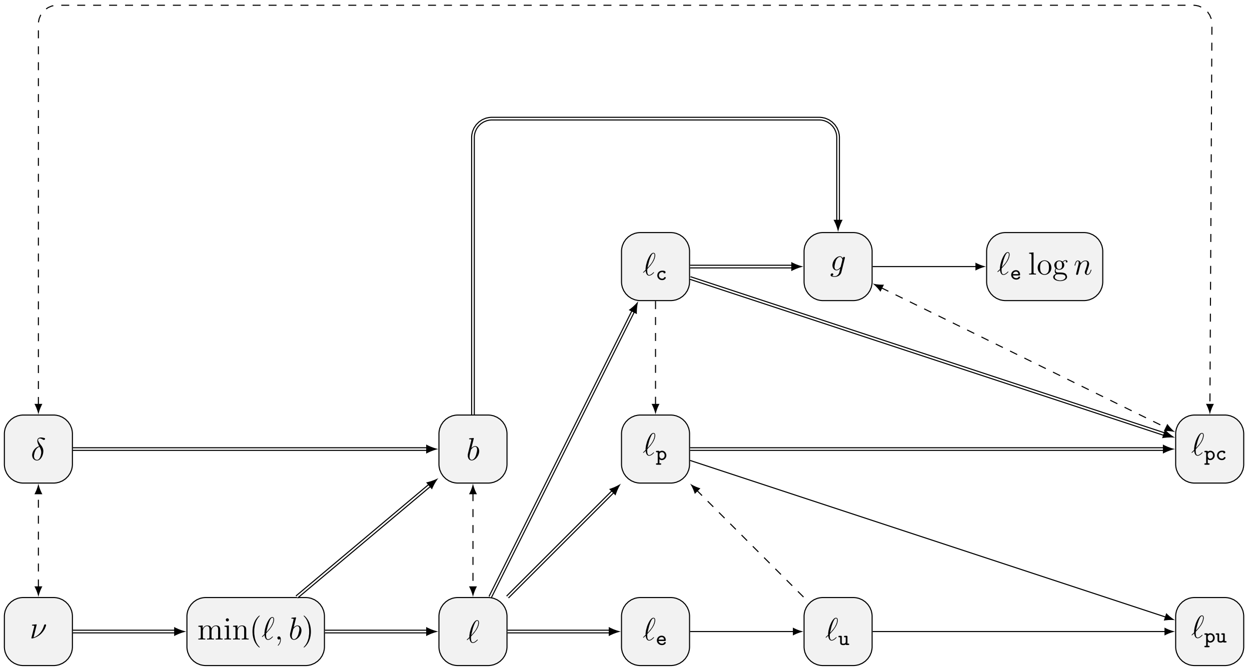


Figure 6.3: Asymptotic relations between the measure ℓ and its variants, the measure ν, and other relevant state of the art repetitiveness measures. A solid arrow from a measure v_1 to a measure v_2 means that it always holds that $v_1 = \mathcal{O}(v_2)$. A double solid arrow from v_1 to v_2 means that it also exists a string family where $v_1 = o(v_2)$. A dashed arrow from v_1 to v_2 means that there exists a family where $v_1 = o(v_2)$.

Chapter 7

Extending Grammar-Based Measures

In Chapter 6, we introduced the measures ℓ and ν, based on L-systems and NU-systems respectively, which can be considerably smaller than δ in some relevant string families. This fact questions if δ should be considered as a golden measure for repetitiveness. On the other hand, providing efficient direct access in $\mathcal{O}(\nu)$ or even $\mathcal{O}(\ell)$ space has been an elusive task. In the case of L-systems, the provided methods for direct access depend on the depth of the system, which can be $\mathcal{O}(n)$. It is possible to provide efficient direct access in $\mathcal{O}(\ell_{\mathtt{e}} \log n)$ space, though it does not break δ.

In this chapter, we go further and show how to extend straight-line programs in order to obtain compression devices defining reachable measures that can break the δ lower bound, while also providing efficient poly-logarithmic time access to arbitrary positions of the text in competitive space (smaller than $\mathcal{O}(g_{\mathtt{rl}})$).

This chapter is structured as follows.

- In Section 7.1, we extend a famous result of Ganardi et al. [52], which shows that any SLP of size g generating a text of length n can be balanced, i.e., we can produce another SLP of size $\mathcal{O}(g)$ whose derivation tree is of height $\mathcal{O}(\log n)$. Our extension is called *Generalized SLPs (GSLPs)*, which allow rules of the form $A \rightarrow x$ (of size $|x|$), where x is a *program* (in any Turing-complete formalism) that outputs the right-hand side of the rule (a string of non-terminals). We show that, if every non-terminal appearing in x's output occurs at least twice, then the GSLP can be balanced in the same way as SLPs.
- In Section 7.2, we explore a particular case of GSLP we call *Iterated SLPs (ISLPs)*. ISLPs extend SLPs (and RLSLPs) by allowing a more complex version of the rule $A \rightarrow B^t$, namely $A \rightarrow \Pi_{i=k_1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$, of size $2 + 2t$. We show that some text families are generated by an ISLP of size $\mathcal{O}(\delta/\sqrt{n})$, thereby sharply breaking the $\Omega(\delta)$ barrier.
- In Section 7.3, using the fact that ISLPs are GSLPs and thus can be balanced, we show how to extract a substring of length λ from the ISLP in time $\mathcal{O}(\lambda + \log^2 n \log\log n)$, as well as computing substring queries like range minimum and next/previous smaller

value, in time $\mathcal{O}(\log^2 n \log\log n)$.

- Finally, in Section 7.4, we apply the balancing result to RLSLPs, which allow rules of the form $A \to B^t$. While the results on ISLPs are directly inherited (because RLSLPs are ISLPs) with the polylogs becoming just the nearly-optimal $\mathcal{O}(\log n)$ [128], we give a general technique to compute a wide family of "composable" queries f on substrings (i.e., $f(X \cdot Y)$ can be computed from $f(X)$ and $f(Y)$).

7.1 Generalized SLPs and How to Balance Them

We introduce a new class of SLP which we show can be balanced so that its derivation tree is of height $\mathcal{O}(\log n)$.

Definition 7.1.1 A *generalized straight-line program* (GSLP) is an SLP that allows special rules of the form $A \to x$, where x is a *program* (in any Turing-complete language) of length $|x|$ whose output $\texttt{OUT}(x)$ is a nonempty sequence of variables, none of which can reach A. The rule $A \to x$ contributes $|x|$ to the size of the GSLP; the standard SLP rules contribute as usual. A special rule $A \to x$ is said to be *balanceable* if every variable occurring in $\texttt{OUT}(x)$ appears at least twice on it. A GSLP is said to be *balanceable* if all its special rules are balanceable.

We can choose any desired language to describe the programs x. Though in principle $|x|$ can be taken as the Kolmogorov complexity of $\texttt{OUT}(x)$, we will focus on very simple programs and on the asymptotic value of $|x|$.

We will prove that any balanceable GSLP can be balanced without increasing its asymptotic size. Our proof generalizes that of Ganardi et al. [52, Thm. 1.2] for SLPs in a similar way to how it was extended to balance RLSLPs [103]. Just as Ganardi et al., in this section we will allow SLPs to have rules of the form $A \to B_1 \cdots B_t$, of size t, where each B_j is a terminal or a nonterminal; this can be converted into a strict SLP of the same asymptotic size, incurring only in an additive $\mathcal{O}(\log n)$ increase in height.

We introduce now some definitions and state some results, from the work of Ganardi et al. [52], that we need in order to prove our balancing result for GSLPs.

A *directed acyclic graph* (DAG) is a directed multigraph $D = (V, E)$ without cycles (nor loops). We denote by $|D|$ the number of edges in this DAG. For our purposes, we assume that any DAG has a distinguished node r called the *root*, satisfying that any other node can be reached from r and r has no incoming edges. We also assume that if a node has k outgoing edges, they are numbered from 1 to k, so edges are of the form (u, i, v). The *sink nodes* of a DAG are the nodes without outgoing edges. The set of sink nodes of D is denoted by W. We denote the number of paths from u to v as $\pi(u, v)$, and $\pi(u, V) = \sum_{v \in V} \pi(u, v)$ for a set V of nodes. The number of paths from the root to the sink nodes is $n(D) = \pi(r, W)$.

One can interpret an SLP G generating a string T as a DAG D: There is a node for each variable in the SLP, the root node is the initial variable, variables of the form $A \to a$ are the sink nodes, and a variable with rule $A \to B_1 B_2 \cdots B_t$ has outgoing edges (A, i, B_i) for $i \in [1\,..\,t]$. Note that if D is a DAG representing G, then $n(D) = |\mathtt{exp}(G)| = |T|$.

Definition 7.1.2 (Ganardi et al. [52, page 5]) Let $D = (V, E)$ be a DAG, and define the pairs $\lambda(v) = (\lfloor \log_2 \pi(r, v) \rfloor, \lfloor \log_2 \pi(v, W)) \rfloor)$ for every $v \in V$. The *symmetric centroid decomposition (SC-decomposition)* of a DAG D produces a set of edges between connected nodes with the same λ pairs defined as $E_{\mathtt{scd}}(D) = \{(u, i, v) \in E \mid \lambda(u) = \lambda(v)\}$, partitioning D into disjoint paths called *SC-paths* (some of them possibly of length 0).

The set $E_{\mathtt{scd}}$ can be computed in $\mathcal{O}(|D|)$ time. If D is the DAG of an SLP G, then $|D|$ is $O(|G|)$. The following lemma justifies the name "SC-paths".

Lemma 7.1.3 (Ganardi et al. [52, Lemma 2.1]) Let $D = (V, E)$ be a DAG. Then every node has at most one outgoing and at most one incoming edge from $E_{\mathtt{scd}}(D)$. Furthermore, every path from the root r to a sink node contains at most $2 \log_2 n(D)$ edges that do not belong to $E_{\mathtt{scd}}(D)$.

Note that the sum of the lengths of all SC-paths is at most the number of nodes of the DAG, or equivalently, the number of variables of the SLP.

The following definition and technical lemma are needed to construct the building blocks of our balanced GSLPs.

Definition 7.1.4 (Ganardi et al. [52, page 7]) A *weighted string* is a string $T \in \Sigma^*$ equipped with a *weight function* $|| \cdot || : \Sigma \to \mathbb{N} \backslash \{0\}$, which is extended homomorphically. If A is a variable in an SLP G, then we write $||A||$ for the weight of the string $\mathtt{exp}(A)$ derived from A.

Lemma 7.1.5 (Ganardi et al. [52, Proposition 2.2]) For every non-empty weighted string T of length n one can construct in linear time an SLP G generating T with the following properties:

- G contains at most $3n$ variables.
- All right-hand sides of G have length at most 4.
- G contains suffix variables[1] $S_1, \ldots, S_n$ producing all non-empty suffixes of T.
- every path from S_i to some terminal symbol a in the derivation tree of G has length at most $3 + 2(\log_2 ||S_i|| - \log_2 ||a||)$.

With this machinery, we are ready to prove the main result of this section. Note that we require that GSLP's special rules are always the endpoint of some SC-path, which makes the argument of Ganardi et al. [52] for regular SLPs easily applicable to GSLPs: the balancing procedure does not involve the special variables of the GSLPs.

Theorem 7.1.6 Given a balanceable GSLP G generating a string T, it is possible to construct an equivalent GSLP G' of size $\mathcal{O}(|G|)$ and height $\mathcal{O}(\log n)$ in $\mathcal{O}(|G| + t(G))$ time, where

[1] Namely, $\mathtt{exp}(S_i) = T[i..n]$, for $i \in [1..n]$.

$t(G)$ is the time needed to compute the lengths of the expansion of each variable in G.

Proof. Transform the GSLP G into an SLP H by (conceptually) replacing their special rules $A \to x$ by $A \to \mathtt{OUT}(x)$, and then obtain the SC-decomposition $E_{\mathtt{scd}}(D)$ of the DAG D of H. Observe that the SC-paths of H use the same variables of G, so it holds that the sum of the lengths of all the SC-paths of H is less than the number of variables of G. Also, note that any special variable $A \to x$ of G is necessarily the endpoint (i.e., the last node of a directed path) of an SC-path in D. To see this note that $\lambda(A) \neq \lambda(B)$ for any B that appears in $\mathtt{OUT}(x)$, because $\log_2 \pi(A, W) \geq \log_2(|\mathtt{OUT}(x)|_B \cdot \pi(B, W)) \geq 1 + \log_2 \pi(B, W)$, where $|\mathtt{OUT}(x)|_B$ is the number of occurrences of B within $\mathtt{OUT}(x)$—so $|\mathtt{OUT}(x)|_B \geq 2$ because G is balanceable. This implies that the balancing procedure of Ganardi et al. on H, which transforms the rules of variables that are not the endpoint of an SC-path in the DAG D, will not touch variables that were originally special variables in G.

Let $\rho = (A_0, d_0, A_1), (A_1, d_1, A_2), \ldots, (A_{p-1}, d_{p-1}, A_p)$ be an SC-path of D. It holds that for each A_i with $i \in [0\mathinner{.\,.}p-1]$, in the SLP H its rule goes to two distinct variables, one to the left and one to the right. Thus, for each variable A_i, with $i \in [0\mathinner{.\,.}p-1]$, there is a variable A'_{i+1} that is not part of the path. Let $A'_1 A'_2 \cdots A'_p$ be the sequence of these variables. Let $L = L_1 L_2 \cdots L_s$ be the subsequence of left variables of the previous sequence. Then construct an SLP of size $\mathcal{O}(s) \subseteq \mathcal{O}(p)$ for the sequence L (seen as a string) as in Lemma 7.1.5, using $|\mathtt{exp}(L_i)|$ in H as the weight function. In this SLP, any path from the suffix nonterminal S_i to a variable L_j has length at most $3 + 2(\log_2 ||S_i|| - \log_2 ||L_j||)$. Similarly, construct an SLP of size $\mathcal{O}(t) \subseteq \mathcal{O}(p)$ for the sequence $R = R_1 R_2 \cdots R_t$ of right symbols in reverse order, as in Lemma 7.1.5, but with prefix variables P_i instead of suffix variables. Each variable A_i, with $i \in [0\mathinner{.\,.}p-1]$, derives the same string as $w_l A_p w_r$, for some suffix w_l of L and some prefix w_r of R. We can find rules deriving these prefixes and suffixes in the SLPs produced in the previous step, so for any variable A_i, we construct an equivalent rule of length at most 3. Add these equivalent rules, and the left and right SLP rules to a new GSLP G'. Do this for all SC-paths. Finally, add the original terminal variables and special variables (which are left unmodified) of the GSLP G, so G' is a GSLP equivalent to G.

Figure 7.1 shows an example where the special GSLP rules are of the form $A \to B^t$, meaning t copies of B (i.e., the GSLP is an RLSLP).

The SLP constructed for L has all its rules of length at most 4, and $3s \leq 3p$ variables. The same happens with R. The other constructed rules also have a length of at most 3, and there are p of them. Summing over all SC-paths, we have $\mathcal{O}(|G|)$ size. The special variables cannot sum up to more than $\mathcal{O}(|G|)$ size. Thus, the GSLP G' has size $\mathcal{O}(|G|)$.

Any path in the derivation tree of G' is of length $\mathcal{O}(\log n)$. To see why, let $A_0, \ldots, A_p$ be an SC-path. Consider a path from a variable A_i with $i \in [0, p]$, to the occurrence of some variable within the string of variables produced by the right-hand side of A_p in G'. Clearly, this path has length at most 2 (i.e., from A_i to A_p and from A_p to such variable). Now consider a path from A_i to a variable A'_j in the sequence L of left variables of the SC-path, with $i < j \leq p$ (that is, A'_j is a variable that diverges from the SC-path before reaching A_p). By construction this path is of the form $A_i \to S_k \to^* A'_j$ (where $\to^*$ represents a path) for some suffix variable S_k (if the occurrence of A'_j is a left symbol), and per Lemma 7.1.5 its length is at most $1+3+2(\log_2 ||S_k|| - \log_2 ||A'_j||) \leq 4 + 2\log_2 ||A_i|| - 2\log_2 ||A'_j||$. Analogously,

$$\begin{array}{lll}
A_0 & \to & A_1 A_{12} \\
A_1 & \to & A_{11} A_2 \\
A_2 & \to & A_5 A_3 \\
A_3 & \to & A_4 A_6 \\
A_4 & \to & A_5^5 \\
A_5 & \to & A_{11} A_6 \\
A_6 & \to & A_7 A_{12} \\
A_7 & \to & A_8 A_{12} \\
A_8 & \to & A_{10} A_9 \\
A_9 & \to & A_{10}^5 \\
A_{10} & \to & A_{11} A_{12} \\
A_{11} & \to & \mathtt{0} \\
A_{12} & \to & \mathtt{1}
\end{array}$$

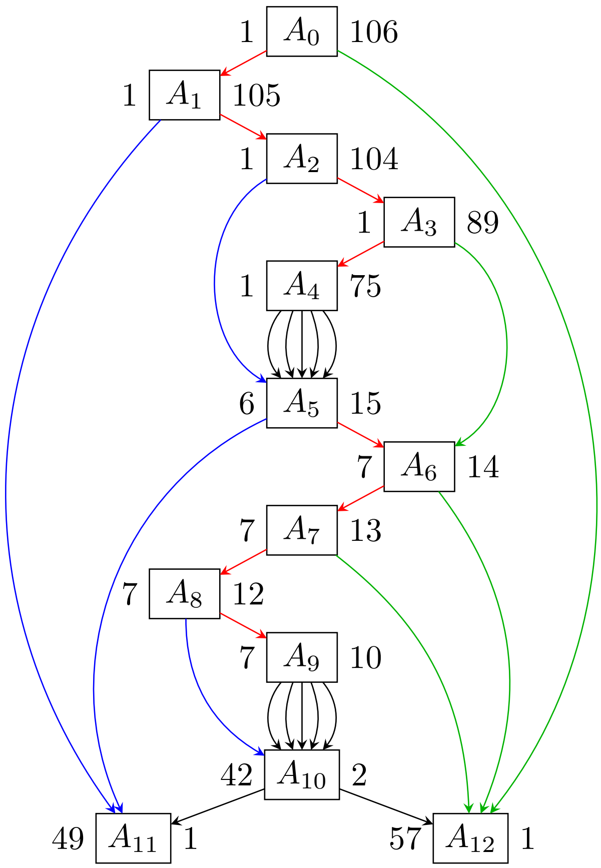


Figure 7.1: The DAG and SC-decomposition of an unfolded RLSLP generating the string $\mathtt{0}(\mathtt{0}(\mathtt{01})^6\mathtt{1}^2)^6(\mathtt{01})^5\mathtt{1}^3$. The value to the left of a node is the number of paths from the root to that node, and the value to the right is the number of paths from the node to sink nodes. Red edges belong to the SC-decomposition of the DAG. Blue (resp. green) edges branch from an SC-path to the left (resp. to the right).

if A'_j is a right variable, the length of the path is bounded by $1+3+2(\log_2||P_k||-\log_2||A'_j||) \leq 4+2\log_2||A_i||-2\log_2||A'_j||$ (where P_k is some prefix variable). We call all these paths whose length we bounded *weight-balanced paths.*

Now consider a maximal path from the root to a leaf in the derivation tree of G'. Factorize it as

$$A_0 \to^* A_1 \to^* \cdots \to^* A_k$$

where each A_i is a variable of H (and also of G and D), and in between each A_i and A_{i+1}, in the DAG D there is almost an SC-path, except that the last edge is not in $E_{\mathtt{scd}}$. Each path $A_i \to^* A_{i+1}$ is a weight-balanced path in the constructed GSLP G'. Simply put, in G', either the path goes directly from A_i to the SC-path endpoint and follows an edge from there to A_{i+1}, or it goes through a suffix (or prefix) variable. In either case, the length of these paths is bounded by 2 or by $4+2\log_2||A_i||-2\log_2||A_{i+1}||$), respectively.

The length of the full path from root to leaf in G' is then at most

$$\sum_{i=0}^{k-1}(4+2\log_2||A_i||-2\log_2||A_{i+1}||) \leq 4k+2\log_2||A_0||-2\log_2||A_k||$$

By Lemma 7.1.3, $k \leq 2\log_2 n$, which yields the upper bound $\mathcal{O}(\log n)$.

To have standard SLP rules of size at most two, delete rules in G' of the form $A \to B$ (replacing all A's by B's), and note that rules of the form $A \to BCDE$ or $A \to BCD$ can be decomposed into rules of length 2, with only a constant increase in size and depth.

The balancing procedure uses $\mathcal{O}(|G| + t(G))$ time and $\mathcal{O}(|G| + s(G))$ auxiliary space, where $t(G)$ and $s(G)$ are the time and space needed to compute and store the set of all the pairs $(B, |\mathtt{OUT}(x)|_B)$, where B appears $|\mathtt{OUT}(x)|_B > 0$ times in $\mathtt{OUT}(x)$, for every special variable $A \to x$. With this information the set $E_{\mathtt{scd}}(D)$ can be computed in $\mathcal{O}(|G| + t(G))$ time, instead of $\mathcal{O}(|H|)$ time. The SLPs of Lemma 7.1.5 are constructed in linear time in the lengths of the SC-paths, which sum to $\mathcal{O}(|G|)$ in total. □

7.2 Iterated Straight-Line Programs

We now define iterated SLPs and show that they can be much smaller than δ.

Definition 7.2.1 An *iterated straight-line program* of *degree* d (d-ISLP) is an SLP that allows in addition *iteration rules* of the form

$$A \to \prod_{i=k_1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$$

where $1 \leq k_1, k_2$, $0 \leq c_1, \ldots, c_t \leq d$ are integers, $B_1, \ldots, B_t$ are variables that cannot reach A (so the ISLP generates a unique string), and the product of strings refers to their concatenation. Iteration rules have size $2 + 2t = \mathcal{O}(t)$ and expand to

$$\mathtt{exp}(A) = \prod_{i=k_1}^{k_2} \mathtt{exp}(B_1)^{i^{c_1}} \cdots \mathtt{exp}(B_t)^{i^{c_t}}$$

where if $k_1 > k_2$ the iteration goes from $i = k_1$ downwards to $i = k_2$. The size $\mathtt{size}(G)$ of a d-ISLP G is the sum of the sizes of all of its rules.

Definition 7.2.2 The measure $g_{\mathtt{it}(d)}(T)$ is defined as the size of the smallest d-ISLP that generates T, whereas $g_{\mathtt{it}}(T) = \min_{d \geq 0} g_{\mathtt{it}(d)}(T)$.

The following observations show that ISLPs subsume RLSLPs, and thus, can be smaller than the smallest L-system.

Proposition 7.2.3 For any $d \geq 0$, it always holds that $g_{\mathtt{it}(d)} = \mathcal{O}(g_{\mathtt{rl}})$.

Proof. Just note that a rule $A \to \prod_{i=1}^{t} B^{i^0}$ from an ISLP simulates a rule $A \to B^t$ from a RLSLP. In particular, 0-ISLPs are equivalent to RLSLPs. □

Proposition 7.2.4 For any $d \geq 0$, there exists a string family where $g_{\mathtt{it}(d)} = o(\ell)$.

Proof. We shown in Chapter 6 a family with $g_{\texttt{rl}} = o(\ell)$ [106]. Hence, $g_{\texttt{it}(d)}$ is also $o(\ell)$ in this family. □

We now show that $d = 1$ suffices to obtain ISLPs that are significantly smaller than δ for some string families.

Lemma 7.2.5 Let $d \geq 1$. There exists a string family with $g_{\texttt{it}(d)} = \mathcal{O}(1)$ and $\delta = \Omega(\sqrt{n})$.

Proof. Such a family is formed by the strings $s_k = \prod_{i=1}^{k} \texttt{a}^i\texttt{b}$. The 1-ISLPs with initial rule $S_k \rightarrow \prod_{i=1}^{k} A^i B$, and rules $A \rightarrow \texttt{a}$, $B \rightarrow \texttt{b}$, generate each string s_k in the family using $\mathcal{O}(1)$ space. On the other hand, it holds that $\delta = \Omega(\sqrt{n})$ in the family $\texttt{c}s_k$ of Chapter 6. As δ can only decrease by 1 after the deletion of a character [1], $\delta = \Omega(\sqrt{n})$ in the family s_k too. □

On the other hand, ISLPs can perform worse than other compressed representations; recall that $\delta \leq \gamma \leq b \leq r/r_{\$}$.

Lemma 7.2.6 Let $\mu \in \{r, r_{\$}, \ell\}$. For any $d \geq 0$, there exists a string family with $g_{\texttt{it}(d)} = \Omega(\log n)$ and $\mu = \mathcal{O}(1)$.

Proof. Consider the family of Fibonacci words defined recursively as $F_0 = \texttt{a}$, $F_1 = \texttt{b}$, and $F_{i+2} = F_{i+1}F_i$ for $i \geq 0$. Fibonacci words cannot contain substrings of the form x^4 for any $x \neq \varepsilon$ [65]. Consider an ISLP for a Fibonacci word and a rule of the form $A \rightarrow \prod_{i=k_1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$. Observe that if $c_r \neq 0$ for some r, then $\max(k_1, k_2) < 4$, as otherwise $\texttt{exp}(B_r)^4$ occurs in T. Similarly, if $c_r = 0$ for all r, then $|k_1 - k_2| < 3$, as otherwise $\texttt{exp}(B_1 \cdots B_t)^4$ appears in T. In the latter case, we can rewrite the product with $k_1, k_2 \in [1\,..\,3]$. Therefore, we can unfold the product rule into standard SLP rules of total size at most $9t$ ($3t$ variables raised to at most 3 each because we assumed our word is Fibonacci). Hence, for any d-ISLP G generating a Fibonacci word, there is an SLP G' of size $\mathcal{O}(|G|)$ generating the same string. As $g = \Omega(\log n)$ in every string family, we obtain that $g_{\texttt{it}(d)} = \Omega(\log n)$ in this family too. On the other hand, $r_{\$}, r$, and ℓ are $\mathcal{O}(1)$ in the even Fibonacci words [102, 95, 105]. □

Lemma 7.2.7 For any $d \geq 0$, there exists a string family satisfying that $z = \mathcal{O}(\log n)$ and $g_{\texttt{it}(d)} = \Omega(\log^2 n / \log\log n)$.

Proof. Let $T(n)$ be the length n prefix of the infinite Thue-Morse word[2] [3] on the alphabet $\{\texttt{a}, \texttt{b}\}$. Let $k_1, ..., k_p$ be a set of distinct positive integers, and consider strings of the form $S = T(k_1)|_1 T(k_2)|_2 \cdots T(k_{p-1})|_{p-1} T(k_p)$, where $|_i$'s are unique separators and k_1 is the largest of the k_i. Since the sequences $T(k_i)$ are cube-free[3] [3], there is no asymptotic difference in the size of the smallest SLP and the smallest ISLP (similarly to Lemma 7.2.6) for the string S. Hence, $g_{\texttt{it}(d)} = \Theta(g)$ in this family. It has been proved that $g = \Omega(\log^2 k_1 / \log\log k_1)$ and $z = \mathcal{O}(\log k_1)$ for some specific sets of integers where $p = \Theta(k_1)$ [16]. Thus, the result follows. □

[2] This is the binary infinite sequence obtained by starting with `0` and appending the binary complement of the string obtained so far, that is, `0 1 10 1001 10010110`...

[3] Those are the sequences that do not contain three consecutive identical substrings.

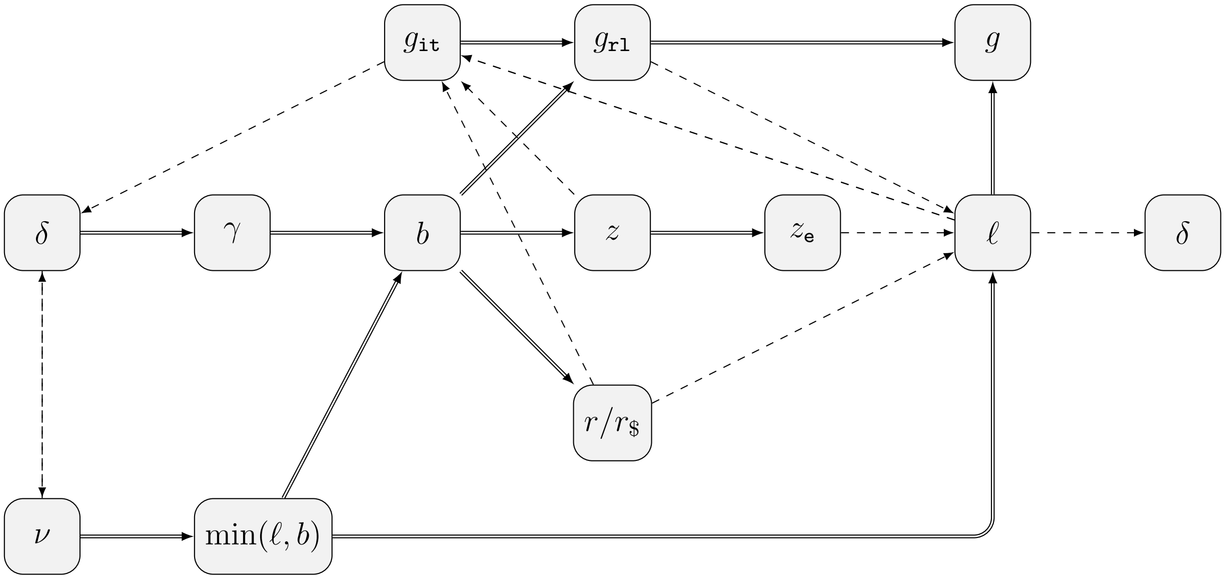


Figure 7.2: Asymptotic relations between ℓ, ν, $g_{\mathtt{it}}$ and other repetitiveness measures. A double solid arrow from v_1 to v_2 means that it always holds that $v_1 = \mathcal{O}(v_2)$, and there exists a string family where $v_1 = o(v_2)$. A dashed arrow from v_1 to v_2 means that there exists a family where $v_1 = o(v_2)$. We suggest the reader to check Figure 4.3 for further implications.

We summarize in Figure 7.2 how $g_{\mathtt{it}}$ is related to other repetitiveness measures.

One thing that makes ISLPs robust is that they are not very sensitive to reversals, morphism application, or edit operations (insertions, deletions, and substitutions of a single character). The measure $g_{\mathtt{it}(d)}$ behaves similarly to SLPs in this matter, for which it has been proved that $g(T') \leq 2g(T)$ after an edit operation that converts T to T' [1], and that $g(\varphi(T)) \leq g(T) + c_\varphi$ with c_φ a constant depending only on the morphism φ [42]. This makes $g_{\mathtt{it}(d)}$ much more robust to string operations than measures like r and $r_\$$, which are highly sensitive to all these transformations [59, 60, 42, 1].

Lemma 7.2.8 Let G be a d-ISLP generating T. Then there exists a d-ISLP of size $|G|$ generating the reversed text T^R. Let φ be a morphism. Then there exists a d-ISLP of size $|G| + c_\varphi$ generating the text $\varphi(T)$, where c_φ is a constant depending only on φ. Moreover, there exists a d-ISLP of size at most $\mathcal{O}(|G|)$ generating T' where T and T' differ by one edit operation.

Proof. For the first claim, note that reversing all the SLP rules and expressions inside the special rules, and swapping the values k_1 and k_2 in each special rule is enough to obtain a d-ISLP of the same size generating T^R.

For the second claim, we replace rules of the form $A \to \mathtt{a}$ with $A \to \varphi(\mathtt{a})$, yielding a grammar of size less than $|G| + \sum_{a \in \Sigma} |\varphi(a)|$. Then we replace these rules with binary rules, which asymptotically do not increase the size of the grammar.

For the edit operations, we proceed as follows. Consider the derivation tree of the ISLP, and the path from the root to the character we want to substitute, delete, or insert a character before or after. Then, we follow this path in a bottom up manner, constructing a new variable A' for each node A we visit. We start at some $A \rightarrow \mathtt{a}$, so we construct $A' \rightarrow x$ where either $x = \mathtt{c}$ or $x = \mathtt{ac}$ or $x = \mathtt{ca}$ or $x = \varepsilon$ depending on the edit operation. If we reach a node $A \rightarrow BC$ going up from B (so we already constructed B'), we construct a node $A' \rightarrow B'C$ (analogously if we come from C). If we reach a node $A \rightarrow \prod_{i=k_1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$ going up from a specific B_r with $r \in [1\,..\,t]$ (so we already constructed B_r') at the k-th iteration of the product with $k_1 \leq k \leq k_2$ and being the q-th copy of B_r inside $B_r^{k^{c_r}}$, then we construct the following new rules

$$
\begin{aligned}
&A_1 \rightarrow \prod_{i=k_1}^{k-1} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}},\ A_2 \rightarrow \prod_{i=k}^{k} B_1^{i^{c_1}} \cdots B_{r-1}^{i^{c_{r-1}}},\ A_3 \rightarrow \prod_{i=1}^{q-1} B_r^{i^0},\\
&A_4 \rightarrow \prod_{i=q+1}^{k^{c_r}} B_r^{i^0},\ A_5 \rightarrow \prod_{i=k}^{k} B_{r+1}^{i^{c_{r+1}}} \cdots B_t^{i^{c_t}},\ A_6 \rightarrow \prod_{i=k+1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}\\
&A' \rightarrow A_1 A_2 A_3 B_r' A_4 A_5 A_6
\end{aligned}
$$

which are equivalent to A (except by the modified, inserted, or deleted symbol) and sum to a total size of at most $6t + 21$. As $t \geq 1$, it holds that $(6t + 21)/(2t + 2) \leq 7$. After finishing the whole process, we obtain a d-ISLP of size at most $8|G|$. Note that this ISLP contains ε-rules. It also contains some non-binary SLP rules, which can be transformed into binary rules, at most doubling the size of the grammar. □

7.3 Accessing ISLPs

We have shown that $g_{\mathtt{it}(d)}$ breaks the lower bound δ already for $d \geq 1$. We now show that the measure is accessible. Concretely, we will prove the following result along Sections 7.3.2 to 7.3.4. Before proving it, Section 7.3.1 shows how the result can be specialized by properly bounding h and d. At the end, Section 7.3.5 extends the result to computing functions over substrings, without need of extracting them first.

Theorem 7.3.1 Let $T[1\,..\,n]$ be generated by a d-ISLP G of height h. Then, we can build in time $\mathcal{O}((|G| + d)d\lceil d \log d/ \log n\rceil)$ and space $\mathcal{O}(|G| + d\lceil d \log d/ \log n\rceil)$ a data structure of size $\mathcal{O}(|G|)$ that extracts any substring of T of length λ in time $\mathcal{O}(\lambda + (h + \log n + d)d\lceil d \log d/ \log n\rceil)$ on a RAM machine of $\Theta(\log n)$ bits, using $\mathcal{O}(h + d\lceil d \log d/ \log n\rceil)$ additional words of working space.

7.3.1 Specializing the result

Before proving Theorem 7.3.1, we obtain a useful special case by showing that both h and d can be bounded to $\mathcal{O}(\log n)$ without increasing the asymptotic size of G. Theorem 7.3.1 then implies the following result.

Theorem 7.3.2 Let $T[1\,..\,n]$ be generated by an ISLP G. Then, we can build in time $\mathcal{O}((|G|+\log n) \log n \log \log n)$ and space $\mathcal{O}(|G|+\log n \log \log n)$ a data structure of size $\mathcal{O}(|G|)$

that extracts any substring of T of length λ in time $\mathcal{O}(\lambda + \log^2 n \log\log n)$ on a RAM machine of $\Theta(\log n)$ bits, using $\mathcal{O}(\log n \log\log n)$ additional words of working space.

To prove that Theorem 7.3.1 implies Theorem 7.3.2, we first show that we can always make $h = \mathcal{O}(\log n)$ without asymptotically increasing the size of the ISLP.

Lemma 7.3.3 Given a d-ISLP G generating a string $T[1\,..\,n]$, it is possible to construct a d'-ISLP G' of size $\mathcal{O}(|G|)$ that generates T, for some $d' \leq d$, with height $h' = \mathcal{O}(\log n)$. The construction requires $\mathcal{O}((|G|+d)d\lceil d\log d/\log n\rceil)$ time and $\mathcal{O}(|G| + d\lceil d\log d/\log n\rceil)$ space.

Proof. ISLPs are GSLPs: they allow rules of the form $A \rightarrow \prod_{i=k_1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$ of size $2+2t$, and a simple program of size $\mathcal{O}(t)$ writes the corresponding right-hand symbols (a sequence over $\{B_1, \ldots, B_t\}$) explicitly. Note that, if $k_1 \neq k_2$ for every special rule, then the corresponding GSLP is balanceable for sure, as no symbol in any output sequence can appear exactly once. If $k_1 = k_2$ for some special rule, instead, the output may have unique symbols $B_j^{i^0}$ or $B_j^{1^{c_j}}$. In this case we can split the rule at those symbols, in order to ensure that they do not appear in special rules, without altering the asymptotic size of the grammar. For example, $A \rightarrow B_1^{i^2} B_2^{i^0} B_3^{i^3} B_4^{i} B_5^{i^0}$ (i.e., $k_1 = k_2 = i$ for some $i > 1$) can be converted into $A \rightarrow A_1 A_2$, $A_1 \rightarrow A_3 B_2$, $A_2 \rightarrow A_4 B_5$, $A_3 \rightarrow B_1^{i^2}$, $A_4 \rightarrow B_3^{i^3} B_4^{i}$. The case $k_1 = k_2 = 1$ corresponds to $A \rightarrow B_1 \cdots B_t$ and can be decomposed into normal binary rules within the same asymptotic size.

We can then apply Theorem 7.1.6. Note the exponents of the special rules $A \rightarrow x$ are retained in general, though some can disappear in the case $k_1 = k_2 = 1$. Thus, the parameter d' of the balanced ISLP satisfies $d' \leq d$.

The time to run the balancing algorithm is linear in $|G|$, except that we need to count, in the rules $A \rightarrow \prod_{i=k_1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$, how many occurrences of each nonterminal are produced. If we define

$$p_c(k) \;=\; \sum_{i=1}^{k} i^c, \tag{7.1}$$

then B_j is produced $p_{c_j}(k_2) - p_{c_j}(k_1 - 1)$ times on the right-hand side of A.

Computing $p_c(k)$ straightforwardly takes time $\Omega(k)$, which may lead to a balancing time proportional to the length n of T. In order to obtain time proportional to the grammar size $|G|$, we need to process the rule for A in time proportional to its size, $\mathcal{O}(t)$. We show next how this can be done, by regarding $p_c(k)$ as a polynomial on k.

Proposition 7.3.4 After a preprocessing time of $\mathcal{O}(d^2\lceil d\log d/\log n\rceil)$, and within $\mathcal{O}(d\lceil d\log d/\log n\rceil)$ working space, we can compute any polynomial $p_c(k)$ in time $\mathcal{O}(d\lceil d\log d/\log n\rceil)$.

Proof. An alternative formula[4] [76] computes $p_c(k)$ using rational arithmetic (note $c \leq d$):

$$p_c(k) \;=\; k^c + \frac{1}{c+1} \cdot \sum_{j=0}^{c} \binom{c+1}{j} b_j \cdot k^{c+1-j}. \qquad (7.2)$$

The formula requires $\mathcal{O}(c) \subseteq \mathcal{O}(d)$ arithmetic operations once the numbers b_j are computed. Those b_j are the Bernoulli (rational) numbers. All the Bernoulli numbers from b_0 to b_d can be computed in $\mathcal{O}(d^2)$ arithmetic operations using the recurrence

$$\sum_{j=0}^{d} \binom{d+1}{j} b_j \;=\; 0,$$

from $b_0 = 1$. The numerators and denominators of the rationals b_j fit in $\mathcal{O}(j \log j) = \mathcal{O}(d \log d)$ bits,[5] so they can be operated in time $\mathcal{O}(\lceil d \log d / \log n \rceil)$ in a RAM machine with word size $\Theta(\log n)$. The total construction time of the Bernoulli numbers is then $\mathcal{O}(d^2 \lceil d \log d / \log n \rceil)$, and they can be maintained in $\mathcal{O}(d \lceil d \log d / \log n \rceil)$ space. □

Therefore, once we build the Bernoulli rationals b_j in advance, in time $\mathcal{O}(d^2 \lceil d \log d / \log n \rceil)$, the processing time for a rule of size $\mathcal{O}(t)$ is $\mathcal{O}(t\, d \lceil d \log d / \log n \rceil)$, which adds up to $\mathcal{O}(|G|\, d \lceil d \log d / \log n \rceil)$ for all the grammar rules. Storing the precomputed values b_j during construction requires $d \lceil d \log d / \log n \rceil$ extra space. □

We now prove that we can always make $d = \mathcal{O}(\log n)$ without changing the size of the ISLP. From now on in the chapter, we will disregard for simplicity the case $k_1 > k_2$ in the rules $A \rightarrow \Pi_{i=k_1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$, as their treatment is analogous to that of the case $k_1 \leq k_2$.

Lemma 7.3.5 If a d-ISLP G generates $T[1\,..\,n]$, then there is also a d'-ISLP G' of the same size that generates T, for some $d' \leq \log_2 n$.

Proof. For any rule $A = \prod_{i=k_1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$, any $i \in [k_1\,..\,k_2]$, and any c_j, it holds that $n \geq |\texttt{exp}(A)| \geq i^{c_j}$, and therefore $c_j \leq \log_i n$, which is bounded by $\log_2 n$ for $i \geq 2$. Therefore, if $k_2 \geq 2$, all the values c_j can be bounded by some $d' \leq \log_2 n$. A rule with $k_1 = k_2 = 1$ is the same as $A \rightarrow B_1 \cdots B_t$, so all values c_j can be set to 0 without changing the size of the rule. □

An even more special case.

The case $d = \mathcal{O}(1)$ deserves to be stated explicitly because it yields near-optimal substring extraction time, and because it already breaks the space lower bound $\Omega(\delta)$. We then plug $d = \mathcal{O}(1)$ and (per Lemma 7.3.3) $h = \mathcal{O}(\log n)$ in Theorem 7.3.1 to obtain the following result.

[4] See Wolfram Mathworld's `https://mathworld.wolfram.com/BernoulliNumber.html`, Eqs. (34) and (47).

[5] See `https://www.bernoulli.org`, sections "Structure of the denominator", "Structure of the nominator", and "Asymptotic formulas".

Corollary 7.3.6 Let $T[1\,..\,n]$ be generated by a d-ISLP G, with $d = \mathcal{O}(1)$. Then, we can build in $\mathcal{O}(|G|)$ time and space a data structure of size $\mathcal{O}(|G|)$ that extracts any substring of T of length λ in time $\mathcal{O}(\lambda + \log n)$ on a RAM machine of $\Theta(\log n)$ bits.

Note that the corollary achieves $\mathcal{O}(\log n)$ access time for a single symbol. Verbin and Yu [128] showed that any data structure using space s to represent $T[1\,..\,n]$ requires time $\Omega(\log^{1-\varepsilon} n / \log s)$ time, for *any* $\varepsilon > 0$. Since even SLPs can use space $s = \mathcal{O}(\log n)$ on some texts, they cannot always offer access time $\mathcal{O}(\log^{1-\varepsilon} n)$ for any constant ε. This restriction applies to even smaller grammars like RLSLPs and d-ISLPs for any d.

7.3.2 Data structures

We now start to prove Theorem 7.3.1. In this subsection we focus on defining proper data structures that let us efficiently compute the length of the expansion of any prefix of the right-hand side of every rule

$$A \to \prod_{i=k_1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}.$$

This will be used in Section 7.3.3 to provide direct access to the content of $\texttt{exp}(A)$. While our problem is easily solved by storing the $(k_2 - k_1 + 1) \cdot t$ cumulative lengths, we cannot afford that space. The challenge is to support these queries within space $\mathcal{O}(t)$, that is, proportional to the size of the rule.

Our key idea is that, though t can be large, there are only $d + 1$ distinct values c_j. We will exploit this because, per Lemma 7.3.5, d can be made $\mathcal{O}(\log n)$.

Navigating within a block

We start focusing on a single "block" $B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$, for fixed i. Our goal here is to efficiently compute the length of the expansion of $B_1^{i^{c_1}} \cdots B_r^{i^{c_r}}$ (i.e., a prefix of the block). Formally, we will compute the function

$$f_r(i) \;=\; \sum_{j=1}^{r} |\texttt{exp}(B_j)| \cdot i^{c_j},$$

for any $r \in [1\,..\,t]$. We now show how to do thisin time $\mathcal{O}(d)$, using $\mathcal{O}(t)$ space for rule A. We use two structures:

- We precompute an array $S_A[1\,..\,t]$ storing cumulative length information, as follows

  $$S_A[r] = \sum_{1 \le j \le r, c_j = c_r} |\texttt{exp}(B_j)|.$$

  That is, $S_A[r]$ adds up the lengths, up to B_r, of the expansions of (only) those symbols that must be multiplied by i^{c_r}.

	1	2	3	4	5	6	7	8	9
S_A	2	3	6	7	14	13	5	3	18
C_A	1	2	1	0	0	1	2	3	0

$$f_{r=8}(i) = 3i^3 + 5i^2 + 13i + 14$$

$$f_{r=9}(i) = 3i^3 + 5i^2 + 13i + 18$$

$$f^+(k) = \tfrac{9}{12}k^4 + \tfrac{38}{12}k^3 + \tfrac{117}{12}k^2 + \tfrac{304}{12}k$$

Figure 7.3: Data structures built for the ISLP rule $A \to \prod_{i=1}^{5} B^i C^{i^2} D^i E E E^i B^{i^2} C^{i^3} D$, with $|\texttt{exp}(B)| = 2$, $|\texttt{exp}(C)| = 3$, $|\texttt{exp}(D)| = 4$, and $|\texttt{exp}(E)| = 7$. On the right we show some of the polynomials that are computed with these data structures.

- A second array, $C_A[1\,..\,t]$, stores the values $c_1, \ldots, c_t$. We preprocess C_A to solve predecessor queries of the form

  $$\texttt{pred}(A, r, c) = \max\{j \leq r,\ C_A[j] = c\},$$

  that is, the latest occurrence of c in C_A to the left of position r, for every $c = 0, \ldots, d$.

To compute $f_r(i)$, we first calculate the values $r_c = \texttt{pred}(A, r, c)$ for all c. We then evaluate $f_r(i)$ in $\mathcal{O}(d)$ time by adding up $S_A[r_c] \cdot i^c$, because $S_A[r_c]$ adds up all those $|\texttt{exp}(B_j)|$, for $j \leq r$, that must be multiplied by i^c in $f_r(i)$.

Example 7.3.7 The left part of Figure 7.3 shows the arrays S_A and C_A of an example rule. The first (B^i), third (D^i), and sixth (E^i) symbols are raised to the power $i = i^1$ (i.e., $c_1 = c_3 = c_6 = 1$). Thus, $S_A[1] = 2 = |\texttt{exp}(B)|$, $S_A[3] = 6 = S_A[1] + |\texttt{exp}(D)|$, and $S_A[6] = 13 = S_A[3] + |\texttt{exp}(E)|$. To compute $f_8(i)$, we will sum the coefficients of i^1, i^2, and i^3. The term to be multiplied by i^1 is $S_A[6]$, because $6 = \texttt{pred}(A, 8, 1)$ is the last position in $[1\,..\,8]$ of a symbol raised to power $i^1 = i$ (it is E^i). In $S_A[6] = 13$ we have the sum of all the lengths that must be multiplied by i. Similarly, in $S_A[\texttt{pred}(A, 8, 2) = 7]$ we have the total length 5 of the rules that must be multiplied by i^2 (which includes C^{i^2} and B^{i^2}), and in $S_A[\texttt{pred}(A, 8, 3) = 8]$ we have the total length 3 of the rules to be multiplied by i^3 (which is just C^{i^3}). We then compute $f_8(i) = 13 \cdot i + 5 \cdot i^2 + 3 \cdot i^3$.

We now show how to build S_A and how to precompute C_A so that the $d + 1$ queries $\texttt{pred}(A, r, c)$ are computed in $\mathcal{O}(d)$ time.

Building S_A. Array S_A is built in time $\mathcal{O}(t)$ once all the lengths $|\texttt{exp}(\cdot)|$ have been computed, by traversing the nonterminals $B_1, \ldots, B_t$ in the rule of A while maintaining in an array $L[B_j]$ the last position of each distinct nonterminal B_j seen so far in the rule. Formally, the invariant is that, once we arrive at B_r, it holds $L[B] = \max\{i < r,\ B_i = B\}$ for all symbols $B \in \{B_1, \ldots, B_{r-1}\}$ (and $L[B] = 0$ if B has not appeared before B_r). We initialize $S_A[0] \leftarrow 0$ and, at step r, we fill $S_A[r] \leftarrow S_A[L[B_r]] + |\texttt{exp}(B_r)|$ and then set $L[B_r] \leftarrow r$ to restore the invariant. Storing L requires $\mathcal{O}(|G|)$ space at construction time; we use lazy initialization to avoid $\mathcal{O}(|G|)$ initialization time.

Algorithm 8 Computing $f_r(i)$ for nonterminal A, in time $\mathcal{O}(d)$

Input : Values i and r, arrays S_A and C_A, and precomputed values $\texttt{pred}(A,(d+1)j,c)$ for every j and c.

Output: The value $f_r(i)$.

```
$j \leftarrow \lceil r/(d+1) \rceil - 1$    // the chunk where $r$ belongs
for $c \leftarrow 0,\ldots,d$ do    // collect last occurrence of each $c$ to the left of the chunk
    $r_c \leftarrow \texttt{pred}(A,(d+1)j,c)$    // this is precomputed
for $k \leftarrow (d+1)j+1,\ldots,r$ do    // update last occurrences within the chunk
    $c \leftarrow C_A[k]$
    $r_c \leftarrow k$
$s \leftarrow 0$    // knowing the last occurrences of each $c$ up to $r$, compute $f_r(i)$
$p \leftarrow 1$
for $c \leftarrow 0,\ldots,d$ do
    $s \leftarrow s + S_A[r_c] \cdot p$
    $p \leftarrow p \cdot i$
return $s$
```

Preprocessing C_A. We preprocess C_A as follows: we cut C_A into chunks of length $d+1$, and for each chunk $C_A[(d+1)\cdot j+1\,..\,(d+1)\cdot(j+1)]$ we store precomputed values $\texttt{pred}(A,(d+1)\cdot j,c)$ for all $c \in \{0,\ldots,d\}$. That is, each chunk stores the predecessor of every c to its left in C_A. Those precomputed values require only $\mathcal{O}(t)$ space because there are $d+1$ of them per chunk. They can be computed in $\mathcal{O}(t)$ time, on a left-to-right traversal of C_A, by using an array $L'[0\,..\,d]$ analogous to L, which at each position records the last ocurrence seen so far of each value $c \in \{0,\ldots,d\}$. The values $L'[0\,..\,d]$ after processing each position $(d+1)\cdot j$ are precisely the values $\texttt{pred}(A,(d+1)\cdot j,c)$ we store with the chunk j.

Once this precomputation is completed, we answer queries as follows. To compute the values $r_c = \texttt{pred}(A,r,c)$ for all c, we find the chunk $j = \lceil r/(d+1) \rceil - 1$ where r belongs, initialize every $r_c = \texttt{pred}(A,(d+1)\cdot j,c)$ for every c (which is stored with the chunk j), and then scan the chunk prefix $C_A[(d+1)\cdot j+1\,..\,r]$ left to right, correcting every $r_c \leftarrow k$ if $c = C_A[k]$, for $k = (d+1)\cdot j+1\,..\,r$.

Algorithm 8 summarizes the whole process to compute $f_r(i)$, and the next lemma summarizes our result.

Lemma 7.3.8 After $\mathcal{O}(|G|)$ precomputation time using $\mathcal{O}(|G|+d)$ working space, we obtain data structures that use $\mathcal{O}(|G|)$ space and can compute any $f_r(i)$ in time $\mathcal{O}(d)$.

Navigating between blocks

We now complete the calculation of the expansion length of any prefix of the rule of A. The following function adds up the expansion lengths of several whole "blocks".

$$f^+(k) \;=\; \sum_{i=k_1}^{k} f_t(i),$$

that is, $f^+(k)$ is the cumulative sum of the length of the whole expressions $B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$ until $i = k$. The problem is, again, that we cannot afford the space of simply storing the $|k_2 - k_1| + 1$ values $f^+(k)$. We will instead compute $f^+(k)$ by reusing the same data structures we already store for $f_r(i)$.

Just as in Algorithm 8, for each $c = 0, \ldots, d$, we compute $t_c = \mathtt{pred}(A, t, c)$ and $s_c = S_A[t_c]$, which is the total expansion length of the symbols that must be multiplied by i^c in the whole rule. We then multiply s_c by the sum of the factors i^c from $i = k_1$ to $i = k$, $s_c \cdot \sum_{i=k_1}^{k} i^c = s_c \cdot (p_c(k) - p_c(k_1 - 1))$, where $p_c(k)$ is defined in Eq. (7.1). Finally, we compute

$$f^+(k) \quad = \quad \sum_{c=0}^{d} s_c \cdot (p_c(k) - p_c(k_1 - 1)).$$

Since $p_c(k)$ is a polynomial on k of maximum degree $c+1$ (see Eq. (7.2)), $f^+(k)$ is a polynomial on k of maximum degree $d + 1$.

Example 7.3.9 Consider the ISLP of Lemma 7.2.5, defined by the rules $S \rightarrow \prod_{i=1}^{k_2} A^i B$, $A \rightarrow \mathtt{a}$, and $B \rightarrow \mathtt{b}$. The polynomials associated with the representation of the rule S are $i^{c_1} = i$ and $i^{c_2} = 1$. Then, we construct the auxiliary polynomials $f_1(i) = |\mathtt{exp}(A)| i^{c_1} = i$ and $f_2(i) = |\mathtt{exp}(A)| i^{c_1} + |\mathtt{exp}(B)| i^{c_2} = i + 1$. Finally, we construct the polynomial $f^+(k) = \sum_{i=1}^{k} f_2(i) = \sum_{i=1}^{k} (i+1) = \frac{1}{2}k^2 + \frac{3}{2}k$. Indeed, our calculation yields $t_0 = 2$ and $t_1 = 1$, $S_A[1] = S_A[2] = 1$, $s_0 = s_1 = 1$, $s_0(p_0(k) - p_0(0)) = k$ and $s_1(p_1(k) - p_0(k)) = \frac{k(k+1)}{2}$, and $f^+(k)$ is then $k + \frac{k(k+1)}{2}$. Figure 7.3 shows a more complex example.

As shown in Proposition 7.3.4, we can compute all the Bernoulli polynomials, and then the coefficients of all the polynomials $p_c(k)$ in time $\mathcal{O}(d^2 \lceil d \log d / \log n \rceil)$. This yields the following result.

Lemma 7.3.10 Once the structures of Lemma 7.3.8 are built, we can build in time $\mathcal{O}(d^2 \lceil d \log d / \log n \rceil)$ additional data structures that use $\mathcal{O}(d \lceil d \log d / \log n \rceil)$ space, which can compute any $f^+(k)$ in time $\mathcal{O}(d \lceil d \log d / \log n \rceil)$.

7.3.3 Direct access

Now that we can efficiently compute the expansion lengths of rule prefixes, we face our simplest query: given the data structures of size $\mathcal{O}(|G|)$ defined in the previous sections, return the symbol $T[l]$ given an index l. Instead of using extra space to store precomputed values, we start the query process by computing all the polynomials $p_c(k)$, which are the same for every rule, in time $\mathcal{O}(d^2 \lceil d \log d / \log n \rceil)$. With those polynomial coefficients precomputed, we can compute any $f^+(k)$, as well as any $f_r(i)$, for any rule in time $\mathcal{O}(d \lceil d \log d / \log n \rceil)$, using Lemmas 7.3.8 and 7.3.10.

For SLPs with derivation tree of height h, the problem is easily solved in $\mathcal{O}(h)$ time by storing the expansion size of every nonterminal, and descending from the root to the corresponding leaf using $|\mathtt{exp}(B)|$ to determine whether to descend to the left or to the right of every rule $A \to BC$. The general idea for d-ISLPs is similar, but now determining which child to follow in repetition rules is more complex.

To access the l-th character of the expansion of $A \to \prod_{i=k_1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$ we first find the value i such that $f^+(i-1) < l \leq f^+(i)$ by using binary search (we let $f^+(i-1) = 0$ when $i = k_1$). Then, we find the value r such that $f_{r-1}(i) < l - f^+(i-1) \leq f_r(i)$ by using binary search on the subindex of the functions (we let $f_{r-1}(i) = 0$ for any i when $r = 1$). We then know that the search follows by B_r, with offset $l - f^+(i-1) - f_{r-1}(i)$ inside $|\mathtt{exp}(B_r)|^{i^{c_r}}$. The offset within B_r is then easily computed with a modulus. Algorithm 9 gives the details, using `succ` to denote the binary search in an ordered set (i.e., $\mathtt{succ}([x_1 \mathinner{..} x_m], l) = j$ iff $x_{j-1} < l \leq x_j$).

We carry out the first binary search so that, for every i we try, if $f^+(i) < l$ we immediately answer $i+1$ if $l \leq f^+(i+1)$; instead, if $l \leq f^+(i)$, we immediately answer i if $f^+(i-1) < l$. As a result, the search area is initially of length $|\mathtt{exp}(A)|$ and, if the answer is i, the search has finished by the time the search area is of length $\leq f^+(i) - f^+(i-1) = f_t(i)$. Thus, there are $\mathcal{O}(1 + \log(|\mathtt{exp}(A)|/f_t(i)))$ binary search steps. The second binary search is modified analogously so that it carries out $\mathcal{O}(1 + \log(f_t(i)/(i^{c_r}|\mathtt{exp}(B_r)|)))$ steps. Summing the costs of both binary searches, and because $i^{c_r} \geq 1$, we have at most $\mathcal{O}(1+\log(|\mathtt{exp}(A)|/|\mathtt{exp}(B_r)|))$ steps. As the search continues by B_r, the sum of binary search steps telescopes to $\mathcal{O}(h+\log n)$ on an ISLP of height h: assume we traverse the ISLP from the initial symbol A_1 to the symbol A_h. The sum of the binary search costs is of the order of

$$\begin{aligned} & (1 + \log(|\mathtt{exp}(A_1)|/|\mathtt{exp}(A_2)|)) + (1 + \log(|\mathtt{exp}(A_2)|/|\mathtt{exp}(A_3)|)) \\ & \quad + \cdots + (1 + \log(|\mathtt{exp}(A_{h-1})|/|\mathtt{exp}(A_h)|)) \\ = \;& h + \log(|\mathtt{exp}(A_1)|/|\mathtt{exp}(A_h)|) \quad \leq \quad h + \log n. \end{aligned}$$

This yields our result for accessing a single symbol.

Lemma 7.3.11 After the construction-time precomputation of Lemma 7.3.8 and the query-time preprocessing $\mathcal{O}(d^2\lceil \log d/\log n\rceil)$ of Lemma 7.3.10, we can access any symbol $T[l]$ in time $\mathcal{O}((h+\log n)\, d\lceil d \log d/\log n\rceil)$.

Example 7.3.12 We show how to access the $\mathtt{b}$ at position 14 of the string $T = \prod_{i=1}^{5} \mathtt{a}^i\mathtt{b}$. Consider the ISLP G and its auxiliary polynomials computed in Example 1. We start by computing $f^+(2) = 5$. As $l > 5$, we go right in the binary search and compute $f^+(4) = 14$. As $l \leq 14$ we go left, compute $f^+(3) = 9$ and find that $i = 4$. Hence, $T[l]$ lies in the expansion of $A^iB = A^4B$ at position $l_1 = l - f^+(i-1) = 5$. Then, we compute $f_1(4) = 4$. As $l_1 > 4$, we turn right and compute $f_2(4) = 5$, finding that $r = 2$. Hence, $T[l]$ lies in the expansion of $B^{i^0} = B^1$ at position $l_2 = l_1 - f_{r-1}(i) = 1$.

7.3.4 Extracting substrings

The last piece for proving Theorem 7.3.1 is to show how to extract substrings from T. Once we have accessed $T[l]$, it is possible to output the substring $T[l \mathinner{..} l+\lambda-1]$ in $\mathcal{O}(\lambda + h)$ additional time, as we return from the recursion in Algorithm 9. We carry the parameter λ

Algorithm 9 Direct access on d-ISLPs of height h in $\mathcal{O}((h + \log n + d)d)$ operations

```
Input  : A variable $A$ of an ISLP, and a position $l \in [1, |\texttt{exp}(A)|]$.
Output: The character $\texttt{exp}(A)[l]$.
function ACCESS$(A, l)$
    if $A \to a$ then      // found the leaf in the parse tree for $T[l]$
        return $a$
    else if $A \to BC$ then      // go left or right as on classic SLPs
        if $l \leq |\texttt{exp}(B)|$ then
            return ACCESS$(B, l)$
        else $(l > |\texttt{exp}(B)|)$
            return ACCESS$(C, l - |\texttt{exp}(B)|)$
    else $(A \to \prod_{i=k_1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}})$      // find the proper descendant node
        $i \leftarrow \texttt{succ}([f^+(k_1) \mathinner{..} f^+(k_2)], l)$
        $l \leftarrow l - f^+(i-1)$
        $r \leftarrow \texttt{succ}([f_1(i) \mathinner{..} f_t(i)], l)$
        $l \leftarrow l - f_{r-1}(i)$
        return ACCESS$(B_r, (l - 1 \bmod |\texttt{exp}(B_r)|) + 1)$
```

of the number of symbols (yet) to output, which is first decremented when we finally find the first symbol, $T[l]$, which we now output immediately. From that point, as we return from the recursion, we output up to λ following symbols and return the number of remaining symbols yet to output, until $\lambda = 0$. See Algorithm 10.

To analyze this algorithm, we note that it visits λ consecutive leaves in the parse tree, plus their ancestors. This is because the algorithm does not visit any node that is not an ancestor of a leaf that must be output: it first traverses towards $T[l]$, and then enters into a node only if there are remaining descendant leaves to visit (i.e., $\lambda > 0$). The ancestors of those leaves are composed of (i) the leftmost and rightmost paths that lead to $T[l]$ and $T[l + \lambda - 1]$, and (ii) a set of complete subtrees between those paths. The former contain up to $2h$ nodes; the latter include up to λ leaves and thus up to λ internal nodes, as there are no nodes of degree 1 in the parse tree.

The analysis also shows up in Algorithm 10. We distinguish two types of recursive calls. Initially the substring to extract is within one of the children of the grammar tree node, and thus only one recursive call is made. Those are the cases of lines 7, 11, and 17. The number of those calls is limited by the the height h of the grammar. Once we reach a node where the substring to extract spreads across more than one child, the λ symbols to output are distributed across more than one recursive call, ending in line 3 when outputting individual symbols. Those recursive calls form a tree with no unary paths and λ leaves, thus they add up to $\mathcal{O}(\lambda)$.

A final detail is that, in line 21 of Algorithm 10, we need to compute i^{c_r}. This can be done with modular exponentiation in time $\mathcal{O}(\log c_r) \subseteq \mathcal{O}(\log d)$. If $\lambda \geq |\texttt{exp}(B_r^{i^{c_r}})|$, then the time $\mathcal{O}(\log c_r)$ to compute i^{c_r} is absorbed by the time to traverse the subtree of $B_r^{i^{c_r}}$.[6] Otherwise, $B_r^{i^{c_r}}$ is the rightmost symbol of the parse tree that we will traverse; this can happen h times

[6] Except if $i = 1$, where the result is simply 1 for any c_r.

only. This issue then adds $\mathcal{O}(h \log d)$ to the total time, which is absorbed by the time to reach $T[l]$; recall Lemma 7.3.11.

The total space for the procedure is $\mathcal{O}(h)$ for the recursion stack (which is unnecessary when returning a single symbol, since recursion can be eliminated in that case), plus $\mathcal{O}(d\lceil d \log d / \log n \rceil)$ for the precomputed Bernoulli rationals.[7] This concludes the proof of Theorem 7.3.1.

7.3.5 Composable functions on substrings

Other than extracting a text substring, we aim at computing more general functions on arbitrary ranges $T[p\,..\,q]$, in time that is independent of the length $q - p + 1$ of the range. We show how to compute some functions that have been studied in the literature, focusing on *composable* ones.

Definition 7.3.13 A function f from strings is *composable* if there exists a function g such that, for every pair of strings X and Y, it holds $f(X \cdot Y) = g(f(X), f(Y))$.

We focus for now on two popular composable functions, which find applications for example on grammar-compressed suffix trees [45, 50].

Definition 7.3.14 A *range minimum query (RMQ)* on $T[p\,..\,q]$ returns the leftmost position where the minimum value occurs in $T[p\,..\,q]$. Formally,

$$\textsc{rmq}(T,p,q) \;=\; \min\{k \in [p\,..\,q] \mid \forall k' \in [p\,..\,q], T[k] \leq T[k']\}.$$

Definition 7.3.15 A *next/previous smaller value query (NSV/PSV)* on $T[p\,..\,n]/T[1\,..\,p]$ and with value v finds the smallest/largest position following/preceding p with value at most v. If there is no such a position, it returns $n+1/\ 0$. Formally,

$$\begin{aligned}\textsc{nsv}(T,p,v) &= \min(\{q \mid q \geq p, T[q] < v\} \cup \{n+1\}),\\ \textsc{psv}(T,p,v) &= \max(\{q \mid q \leq p, T[q] < v\} \cup \{0\}).\end{aligned}$$

We show next how to efficiently solve those queries on ISLPs.

Range Minimum Queries

Solving RMQs on an SLP G is simple thanks to composability. More precisely, what is composable is an extended function $f(X) = \langle m, v, \ell \rangle$ where $m = \textsc{rmq}(X, 1, |X|)$, $v = X[m]$, and $\ell = |X|$. Then, given $f(X) = \langle m_x, v_x, \ell_x \rangle$ and $f(Y) = \langle m_y, v_y, \ell_y \rangle$, it holds $f(X \cdot Y) = \langle m_x, v_x, \ell_x + \ell_y \rangle$ if $v_x \leq v_y$, and $\langle \ell_x + m_y, v_y, \ell_x + \ell_y \rangle$ otherwise, which is computable in time $\mathcal{O}(1)$. We also compute $f(a) = \langle 1, a, 1 \rangle$ in $\mathcal{O}(1)$ time.

[7] As these do not depend on the query, they could be precomputed at indexing time and be made part of the index, at a very modest increase in space.

To compute RMQs on an SLP G, we first preprocess the grammar to store $f(\texttt{exp}(A)) = \langle m, v, \ell \rangle$ for each nonterminal A, in the form of the pair $\texttt{rmq}(A) = \langle m, v \rangle$ and the length $\ell = |\texttt{exp}(A)|$. Thanks to the composability of f, this is easily built in $\mathcal{O}(|G|)$ time in a bottom-up traversal of the grammar.

To solve $\textsc{rmq}(T[p\,..\,q])$ on the SLP, we descend from the root towards $T[p\,..\,q]$ (guided by the stored expansion lengths $|\texttt{exp}(A)|$) until finding a leaf (if $p = q$), or more typically a rule $A \to BC$ such that $T[p\,..\,q] = \texttt{exp}(B)[p'\,..\,|\texttt{exp}(B)|] \cdot \texttt{exp}(C)[1\,..\,q']$. At this point we split into two recursive calls, one computing $\textsc{rmq}$ on a suffix of $\texttt{exp}(B)$ (a *suffix call*) and another on a prefix of $\texttt{exp}(C)$ (a *prefix call*). By making the recursive calls return $\texttt{rmq}(B)$ in $\mathcal{O}(1)$ time when the range spans the whole string $\texttt{exp}(B)$, we ensure that those prefix/suffix calls perform only one further (nontrivial) recursive call, and thus the query is solved in $\mathcal{O}(h)$ time, traversing at most two root-to-leaf paths in the parse tree. Algorithm 11 shows the details.

To solve RMQs on ISLPs, we observe that the expansion of $A \to \prod_{i=k_1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$ always contains the same symbols. Further, the RMQ of $\texttt{exp}(A)$ occurs always in the first block, $i = k_1$, and it depends essentially on the sequence $B_1 \cdots B_t$. To handle these rules, we preprocess them as follows. Let $\texttt{rmq}(B_j) = \langle m_j, v_j \rangle$. Then, we build the string $v_1 \cdots v_t$ and precompute an RMQ data structure on it that answers queries $\texttt{rmq}_A(p, q) = \textsc{rmq}(v_1 \cdots v_t, p, q)$. It is possible to build such a data structure in $\mathcal{O}(t)$ time and bits of space, such that it answers queries in $\mathcal{O}(1)$ time [44], so this adds just $\mathcal{O}(|G|)$ time and bits to the grammar preprocessing cost. With this structure, we can simulate the extension of our $\texttt{rmq}(A)$ precomputed pairs to any subsequence $B_a^{i^{c_a}} \cdots B_b^{i^{c_b}}$ of $B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$: $\texttt{rmq}(B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}, a, b) = \langle m, v \rangle$, where $\textsc{rmq}(v_1 \cdots v_t, a, b) = m'$, $\texttt{rmq}(B_{m'}) = \langle m'', v \rangle$, and $m = f_{m'-1}(i) + m''$. The time to compute this is dominated by the $\mathcal{O}(d)$ cost to compute $f_{m'-1}(i)$.

At query time, when we arrive at such a node A with limits p and q, we proceed as in lines 10–13 of Algorithm 9 to find the values i_p and r_p, and i_q and r_q, corresponding to p and q, respectively (just as we find i and r for l in Algorithm 9). There are several possibilities:

1. If $i_p = i_q$ and $r_p = r_q$, then p and q fall inside $\texttt{exp}(B_{r_p}^{i^{c_{r_p}}})$. They may be both inside a single copy of $\texttt{exp}(B_{r_p})$, in which case we continue with a single recursive call. Or they may span a (possibly empty) suffix of $\texttt{exp}(B_{r_p})$, zero or more copies of $\texttt{exp}(B_{r_p})$, and a (possibly empty) prefix of $\texttt{exp}(B_{r_p})$. The query is then solved with at most two recursive calls on B_{r_p} (which are prefix/suffix calls), and the information on $\texttt{rmq}(B_{r_p})$. We compose as explained those (up to) three results, and add $f^+(i_p - 1) + f_{r_p-1}(i_p)$ to the resulting position so as to place it within $\texttt{exp}(A)$.

2. If $i_p = i_q$ and $r_p < r_q$, then we must also consider the subsequence $B_{r_p+1}^{i_p^{c_{r_p+1}}} \cdots B_{r_q-1}^{i_p^{c_{r_q-1}}}$, in case $r_q - r_p > 1$. This additional candidate to the RMQ is found with $\texttt{rmq}(B_1^{i_p^{c_1}} \cdots B_t^{i_p^{c_t}}, r_p + 1, r_q - 1)$, in time $\mathcal{O}(d)$ as explained.

3. If $i_p < i_q$, we must also add a suffix of of $B_1^{i_p^{c_1}} \cdots B_t^{i_p^{c_t}}$, the whole $B_1^{(i_p+1)^{c_1}} \cdots B_t^{(i_p+1)^{c_t}}$ (if $i_q - i_p > 1$), and a prefix of $B_1^{i_q^{c_1}} \cdots B_t^{i_q^{c_t}}$ (if $i_q - i_p = 1$). All those are included with our simulation of queries $\texttt{rmq}(B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}, a, b)$.

Overall, we perform either one recursive call (when p and q are inside the same B_{r_p}), or two prefix/suffix recursive calls (for a suffix of B_{r_p} and a prefix of B_{r_q}). The analysis is then the same as for the SLPs, yielding time $\mathcal{O}(hd)$. This is in addition to (and dominated by) the $\mathcal{O}((h+\log n)d\lceil d\log d/\log n\rceil)$ time, plus the preprocessing time of $\mathcal{O}(d^2\lceil d\log d/\log n\rceil)$, due to the binary searches needed to find i_p, i_q, r_p, and r_q, as for direct access (recall Lemma 7.3.11).

Theorem 7.3.16 Let $T[1\,..\,n]$ be generated by a d-ISLP G of height h. Then, we can build in time $\mathcal{O}((|G|+d)d\lceil d\log d/\log n\rceil)$ and space $\mathcal{O}(|G|+d\lceil d\log d/\log n\rceil)$ a data structure of size $\mathcal{O}(|G|)$ that computes any query $\textsc{rmq}(T,p,q)$ in time $\mathcal{O}((h+\log n+d)d\lceil d\log d/\log n\rceil)$ on a RAM machine of $\Theta(\log n)$ bits, using $\mathcal{O}(h+d\lceil d\log d/\log n\rceil)$ additional words of working space.

Since we can make both h and d be $\mathcal{O}(\log n)$ per Lemmas 7.3.3 and 7.3.5, we have the following corollary.

Corollary 7.3.17 Let $T[1\,..\,n]$ be generated by an ISLP G. Then, we can build in time $\mathcal{O}((|G|+\log n)\log n\log\log n)$ and space $\mathcal{O}(|G|+\log n\log\log n)$ a data structure of size $\mathcal{O}(|G|)$ that computes any query $\textsc{rmq}(T,p,q)$ in time $\mathcal{O}(\log^2 n\log\log n)$ on a RAM machine of $\Theta(\log n)$ bits, using $\mathcal{O}(\log n\log\log n)$ additional words of working space.

Finally, the following specialization is relevant, as for example it encompasses 1-ISLPs (which may break δ) and RLSLPs, and matches the analogous result on SLPs.

Corollary 7.3.18 Let $T[1\,..\,n]$ be generated by a d-ISLP G with $d=\mathcal{O}(1)$. Then, we can build in time and space $\mathcal{O}(|G|)$ a data structure of size $\mathcal{O}(|G|)$ that computes any query $\textsc{rmq}(T,p,q)$ in time $\mathcal{O}(\log n)$ on a RAM machine of $\Theta(\log n)$ bits.

Next/Previous Smaller Value

Let us consider query NSV; query PSV is analogous. NSV is composable if we extend it to function $f(X,v)=\langle p,\ell\rangle$, where $p=\textsc{nsv}(X,1,v)$ and $\ell=|X|$. If $f(X,v)=\langle p_x,\ell_x\rangle$ and $f(Y,v)=\langle p_y,\ell_y\rangle$, then $f(X\cdot Y)=\langle p,\ell_x+\ell_y\rangle$, where $p=p_x$ if $p_x\le\ell_x$, else $p=\ell_x+p_y$ if $p_y\le\ell_y$, and $p=\ell_x+\ell_y+1$ otherwise. The composition takes $\mathcal{O}(1)$ time.

The procedure to compute $\textsc{nsv}(T,p,v)$ on an SLP is depicted in Algorithm 12. We reuse the precomputed pairs $\texttt{rmq}(A)=\langle m,v\rangle$ of RMQs, using $\texttt{rmq}(A).v$ to refer to v. Importantly, the algorithm uses that field to notice in constant time that the answer is not within $\texttt{exp}(A)$ (lines 2–3). In this case we say that the call to A *fails* (to find the answer within $\texttt{exp}(A)$). As for RMQs, the algorithm may perform two calls on $A\to BC$, which only happens when the call on B fails, but then the call on B is a suffix call and the call on C is a prefix call. Note that, in this asymmetric query with no right limit, a prefix call on C is a call on the whole $\texttt{exp}(C)$; we call it a *whole-symbol call.* As explained, those calls take $\mathcal{O}(1)$ time when they fail. Therefore,

- The suffix call starting from B, which finally fails, cannot branch again into two recursive calls at a symbol $A' \to B'C'$, because once the call on B' fails, the call on C' is a whole-symbol call, and this fails in constant time.

- The prefix call starting from C cannot branch again into two recursive calls at a symbol $A' \to B'C'$, because this occurs only if the call on B' fails. Since this is a whole-symbol call on B', it fails in constant time.

Since at most two paths are followed from the first branching into two calls, the total time is $\mathcal{O}(h)$.

To extend the algorithm to ISLPs we must consider, as for the case of RMQs, the special rules. Just as in that case, the answer to a query $\textsc{nsv}(\texttt{exp}(A), p, v)$ with $A \to \Pi_{i=k_1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$ depends essentially on the smallest values of the nonterminal expansions, $\texttt{exp}(B_j)$. Let again $\texttt{rmq}(B_j) = \langle m_j, v_j \rangle$. We preprocess the string $v_1 \cdots v_t$ to solve queries $\textsc{nsv}(v_1 \cdots v_t, p)$. This preprocessing takes $\mathcal{O}(t \log t)$ time and $\mathcal{O}(t)$ space, and answers NSV queries in time $\mathcal{O}(\log^\varepsilon t)$ for any constant $\varepsilon > 0$ [109] (those are modeled as orthogonal range successor queries on a grid).

We can then simulate precomputed values $\texttt{nsv}(B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}, p, v) = q$, where p refers to $B_p^{i^{c_p}} \cdots B_t^{i^{c_t}}$, with the value $\textsc{nsv}(v_1 \cdots v_t, p, v) = q'$ precomputed as explained, $\textsc{nsv}(B_{q'}, 1, v) = q''$ obtained with a recursive call, and $q = f_{q'-1}(i) + q''$. Note that the recursive call is for a whole symbol, and we are sure to find the answer inside it: if $\textsc{nsv}(v_1 \cdots v_t, p, v) = t + 1$, we return $f_t(i) + 1$ without making any recursive call. At query time, after finding i_p and r_p as for RMQs, we have the following cases:

1. We may have to recurse on a nonempty suffix of B_{r_p}, finishing if we find the answer inside it. If not, there may be more copies of B_{r_p} ahead of position p, in which case we either determine in constant time that there is no answer inside B_{r_p}, or we recurse with a whole-symbol call on B_{r_p} and find the answer inside it, thereby finishing.

2. If not finished, we may have to consider a block suffix $B_{r_p+1}^{i_p^{c_{r_p+1}}} \cdots B_t^{i_p^{c_t}}$. This is handled by computing $\texttt{nsv}(B_1^{i_p^{c_1}} \cdots B_t^{i_p^{c_t}}, r_p+1, v)$ as explained, possibly making a whole-symbol recursive call, only when we are sure to find the answer inside it.

3. If not, we may find the answer in the next block, $B_1^{(i_p+1)^{c_1}} \cdots B_t^{(i_p+1)^{c_t}}$, in the same way as in point 2. If we find no answer here, then there is no answer to NSV and we return $|\texttt{exp}(A)| + 1$.

Just as for the case of SLPs, we traverse only two paths along the process: even if now we have a sequence of more than two symbols (not just $A \to BC$), we are able to determine with a constant amount of $\texttt{nsv}$ queries whether there is an answer to the right of the failing recursive call, and in which symbol we must recurse to find it. The main difference with the cost of RMQs is the $\mathcal{O}(\log^\varepsilon t) \subseteq \mathcal{O}(\log^\varepsilon |G|)$ time incurred to compute $\texttt{nsv}$ queries, and the corresponding $\mathcal{O}(t \log t)$ construction time, which adds up to $\mathcal{O}(|G| \log |G|)$.

Theorem 7.3.19 *Let $T[1\,..\,n]$ be generated by a d-ISLP G of height h. Then, for any constant $\varepsilon > 0$, we can build in time $\mathcal{O}(|G|(\log |G| + d\lceil d \log d / \log n \rceil) + d^2 \lceil d \log d / \log n \rceil)$*

and space $\mathcal{O}(|G| + d\lceil d \log d / \log n \rceil)$ a data structure of size $\mathcal{O}(|G|)$ that computes any query $\textsc{psv}/\textsc{nsv}(T, p, v)$ in time $\mathcal{O}(h \log^{\varepsilon} |G| + (h + \log n + d)d\lceil d \log d / \log n \rceil)$ on a RAM machine of $\Theta(\log n)$ bits, using $\mathcal{O}(h + d\lceil d \log d / \log n \rceil)$ additional words of working space.

Corollary 7.3.20 Let $T[1\,..\,n]$ be generated by an ISLP G. Then, we can build in time $\mathcal{O}((|G|+\log n) \log n \log\log n)$ and space $\mathcal{O}(|G|+\log n \log\log n)$ a data structure of size $\mathcal{O}(|G|)$ that computes any query $\textsc{psv}/\textsc{nsv}(T, p, v)$ in time $\mathcal{O}(\log^2 n \log\log n)$ on a RAM machine of $\Theta(\log n)$ bits, using $\mathcal{O}(\log n \log\log n)$ additional words of working space.

Corollary 7.3.21 Let $T[1\,..\,n]$ be generated by a d-ISLP G with $d = \mathcal{O}(1)$. Then, for any constant $\varepsilon > 0$, we can build in time $\mathcal{O}(|G| \log |G|)$ and space $\mathcal{O}(|G|)$ a data structure of size $\mathcal{O}(|G|)$ that computes any query $\textsc{psv}/\textsc{nsv}(T, p, v)$ in time $\mathcal{O}(\log n \log^{\varepsilon} |G|)$ on a RAM machine of $\Theta(\log n)$ bits.

7.4 Revisiting RLSLPs

As pointed out in Proposition 7.2.3, RSLPs are equivalent to 0-ISLPs, because an ISLP rule $A \rightarrow \prod_{i=k_1}^{k_2} B^{i^0}$ corresponds exactly to the RLSLP rule $A \rightarrow B^{|k_2-k_1|+1}$. We can then apply Lemma 7.3.3 over any RLSLP to obtain an equivalent RLSLP of the same asymptotic size and height $\mathcal{O}(\log n)$. Once we count with a balanced version of any RLSLP, we can reuse Corollaries 7.3.6, 7.3.18, and 7.3.21, to obtain a similar result for RLSLPs. Note that we can improve those results because we do not need to preprocess the grammar to simulate the `rmq` and `nsv` queries on blocks, because in an RLSLP all the cases of run-length rules $A \rightarrow B^t$ fall inside the subcase 1 of RMQs and NSVs.

Corollary 7.4.1 Let $T[1\,..\,n]$ be generated by a RLSLP G. Then, we can build in time and space $\mathcal{O}(|G|)$ data structures of size $\mathcal{O}(|G|)$ that (i) extract any substring $T[l\,..\,l + \lambda - 1]$ in time $\mathcal{O}(\lambda + \log n)$, (ii) compute any query $\textsc{rmq}(T, p, q)$ in time $\mathcal{O}(\log n)$, and (iii) compute any query $\textsc{psv}/\textsc{nsv}(T, p, v)$ in time $\mathcal{O}(\log n)$, on a RAM machine of $\Theta(\log n)$ bits.

Those results on RLSLPs have already been obtained before [50, 32], but our solutions exploiting balancedness are much simpler once projected into the run-length rules. We now exploit the simplicity of RLSLPs to answer a wider range of queries on substrings.

7.4.1 More general functions

We now expand our results to a wide family of composable functions that can be computed in $\mathcal{O}(\log n)$ time on top of balanced RLSLPs. We prove the following result.

Theorem 7.4.2 Let f be a composable function from strings to a set of size $n^{\mathcal{O}(1)}$, computable in time t_f for strings of length 1, with its composing function g being computable in time t_g. Then, given an RLSLP G representing $T[1\,..\,n]$, there is a data structure of size $\mathcal{O}(|G|)$ that can be built in time $\mathcal{O}(|G|(t_f + t_g \log n))$ and that computes any $f(T[i\,..\,j])$ in time $\mathcal{O}(t_g \log n)$.

Proof. By Theorem 7.1.6, we can assume G is balanced. We store the values $L[A] = |\mathtt{exp}(A)|$ and $F[A] = f(\mathtt{exp}(A))$ for every variable A, as arrays. These arrays add only $\mathcal{O}(|G|)$ extra space because the values in F fit in $\mathcal{O}(\log n)$-bit words. Let us overload the notation and use $f(A, i, j) = f(\mathtt{exp}(A)[i\,..\,j])$. Algorithm 13 shows how to compute any $f(A, i, j)$; by calling it on the start symbol S of G we compute $f(T[i\,..\,j]) = f(S, i, j)$.

Just as for ISLPs, in the beginning we follow a single path along the derivation tree, with only one recursive call per argument A (lines 6, 8, and 17). The cost of those calls adds up to the height of the grammar, $\mathcal{O}(\log n)$. This path finishes at a leaf or at an internal node A where $\mathtt{exp}(A)[i\,..\,j]$ spans more than one child of A in the derivation tree, in which case we may perform two recursive calls. Note that in the only places where this may occur (lines 10–11 and 18–19) those recursive calls will be prefix/suffix calls (i.e., either $i = 1$ or $j = L[A]$ when we call $\textsc{f}(A, i, j)$). We now focus on bounding the cost of prefix/suffix calls.

We define $c(A)$ as the highest cost to compute $f(A, i, L[A])$ or $f(A, 1, j)$ over any i and j (i.e., the cost of prefix/suffix calls), charging 1 to the number of calls to function $\textsc{f}$ and t_g to each invocation to function g. We assume for simplicity that $t_g \geq 1$ and prove by induction that $c(A) \leq (1+2t_g)d(A) + 2t_g \log |\mathtt{exp}(A)|$, where $d(A)$ is the distance from A to its deepest descendant leaf in the derivation tree. This certainly holds in the base case of leaves, where $d(A) = 1$; it is included in lines 2–3.

In the inductive case of rules $A \to BC$ (lines 4–12), we note that there can be two calls to $\textsc{f}$, but in prefix/suffix calls one of those calls spans the whole symbol—line 10 in a prefix call or line 11 in a suffix call. Calls that span the whole symbol finish in line 3 and therefore cost just 1. Therefore, we have $c(A) \leq 1 + \max(c(B), c(C)) + t_g$, which by induction is

$$\begin{aligned}
c(A) &\leq 1 + \max(c(B), c(C)) + t_g \\
&\leq 1 + t_g + \max((1{+}2t_g)d(B) + 2t_g \log |\mathtt{exp}(B)|, (1{+}2t_g)d(C) + 2t_g \log |\mathtt{exp}(C)|) \\
&\leq (1 + 2t_g)(1 + \max(d(B), d(C))) + 2t_g \log \max(|\mathtt{exp}(B)|, |\mathtt{exp}(C)|)) \\
&\leq (1 + 2t_g)d(A) + 2t_g \log |\mathtt{exp}(A)|).
\end{aligned}$$

In the inductive case of rules $A \to B^t$ (lines 13–21), a similar situation occurs in lines 18–19: only one of the two recursive calls is nontrivial. Therefore, it holds $c(A) \leq 1 + c(B) + 2t_g \log t + 2t_g$, where the term $2t_g \log t$ comes from the recursive procedure to compute $f_c(t'' - t' - 1)$ in line 20; the logarithm is in base 2. Because $t = |\mathtt{exp}(A)|/|\mathtt{exp}(B)|$, by induction we have

$$\begin{aligned}
c(A) &\leq 1 + c(B) + t_g(2 + 2\log(|\mathtt{exp}(A)|/|\mathtt{exp}(B)|)) \\
&\leq 1 + (1 + 2t_g)d(B) + 2t_g \log |\mathtt{exp}(B)| + 2t_g(1 + \log(|\mathtt{exp}(A)|/|\mathtt{exp}(B)|)) \\
&= (1 + 2t_g)(1 + d(B)) + 2t_g \log |\mathtt{exp}(A)| \;=\; (1 + 2t_g)d(A) + 2t_g \log |\mathtt{exp}(A)|.
\end{aligned}$$

Therefore, the procedure costs $c(A) = (1+2t_g)d(A) + 2t_g \log |\mathtt{exp}(A)| = \mathcal{O}(t_g \cdot \log n)$ from the nonterminal A where the single path splits into two.

Arrays L and F can be precomputed in time $\mathcal{O}(|G|(t_f + t_g \log n))$ via a postorder traversal of the grammar tree. We compute f for every distinct individual symbol and g for each distinct nonterminal A, whose children have by then their L and F entries already computed.

In the case of rules $A \to B^t$, the entry $F[A]$ can be computed in time $\mathcal{O}(t_g \log t)$ with the same mechanism used in line 20 of Algorithm 13. □

We show in the next subsection how to use this result to compute a more complicated function, which in particular we do not know how to compute efficiently on ISLPs.

7.4.2 Application: Karp-Rabin fingerprints

Given a string $T[1\,..\,n]$, a suitable integer c, and a prime number $\mu \in \mathcal{O}(n)$, the Karp-Rabin fingerprint [66] of $T[i\,..\,j]$, for $1 \leq i \leq j \leq n$, is defined as

$$\kappa(T[i\,..\,j]) \;=\; \left(\sum_{k=i}^{j} T[k] \cdot c^{k-i}\right) \bmod \mu.$$

Computation of fingerprints of text substrings from their grammar representation is a key component of various compressed text indexing schemes [32]. While it is known how to compute it in $\mathcal{O}(\log n)$ time using $\mathcal{O}(|G|)$ space on an RLSLP G [32, App. A], we show now a much simpler procedure that is an application of Theorem 7.4.2.

Note that, for any split position $p \in [i\,..\,j-1]$, it holds

$$\kappa(T[i\,..\,j]) \;=\; \left(\kappa(T[i\,..\,p]) + \kappa(T[p+1\,..\,j]) \cdot c^{p-i+1}\right) \bmod \mu. \tag{7.3}$$

We use this property as a basis for the efficient computation of fingerprints on RLSLPs.

Theorem 7.4.3 (cf. [17, 32]) Given an RLSLP G representing $T[1\,..\,n]$ and a Karp-Rabin fingerprint function κ, there is a data structure of size $\mathcal{O}(|G|)$ that can be built in time $\mathcal{O}(|G| \log n)$ and computes fingerprints of arbitrary substrings of T in $\mathcal{O}(\log n)$ time.

Proof. Let $f(X) = \langle \kappa(X), c^{|X|} \rangle$ be the function f to apply Theorem 7.4.2. We then define

$$g(\langle \kappa_x, c_x \rangle, \langle \kappa_y, c_y \rangle) = \langle (\kappa_x + \kappa_y \cdot c_x) \bmod \mu, (c_x \cdot c_y) \bmod \mu \rangle,$$

which can be computed in time $t_g = \mathcal{O}(1)$.

It is easy to see that, by Eq. (7.3), $f(XY) = \langle \kappa(XY), c^{|XY|} \rangle = \langle (\kappa(X) + \kappa(Y) \cdot c^{|X|}) \bmod \mu, (c^{|X|} \cdot c^{|Y|}) \bmod \mu \rangle = g(\langle \kappa(X), c^{|X|} \rangle, \langle \kappa(Y), c^{|Y|} \rangle) = g(f(X), f(Y))$. Therefore, application of Theorem 7.4.2 leads to a procedure that computes $f(T[i\,..\,j]) = \langle \kappa(T[i\,..\,j]), c^{j-i+1} \bmod \mu \rangle$ in time $\mathcal{O}(\log n)$ and using $\mathcal{O}(|G|)$ extra space. □

Algorithm 10 Length-λ substring access on ISLPs of height h in $\mathcal{O}(h+\lambda)$ extra time

Input : A variable A of an ISLP, a position $l \in [1, |\mathtt{exp}(A)|]$ and a length $\lambda > 0$.

Output: Outputs $\mathtt{exp}(A)[l \mathinner{.\,.} l+\lambda-1]$ and returns the number of symbols it could not extract (if $l+\lambda-1 > |\mathtt{exp}(A)|$).

```
function EXTRACT($(A, l, \lambda)$)
    if $A \rightarrow a$ then    // found the leaf in the parse tree for $T[l]$, first output
        output $a$
        $\lambda \leftarrow \lambda - 1$
    else if $A \rightarrow BC$ then    // go left and/or right as in classic SLPs
        if $l \leq |\mathtt{exp}(B)|$ then
            $\lambda \leftarrow$ EXTRACT($((B, l, \lambda))$)
            if $\lambda > 0$ then    // go also right if there are symbols yet to output
                $\lambda \leftarrow$ EXTRACT($(C, 1, \lambda))$)
        else $(l > |\mathtt{exp}(B)|)$
            $\lambda \leftarrow$ EXTRACT($(C, l - |\mathtt{exp}(B)|, \lambda)$)
    else $(A \rightarrow \prod_{i=k_1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}})$    // find the first proper descendant node
        $i \leftarrow \mathtt{succ}([f^+(k_1) \mathinner{.\,.} f^+(k_2)], l)$
        $l \leftarrow l - f^+(i-1)$
        $r \leftarrow \mathtt{succ}([f_1(i) \mathinner{.\,.} f_t(i)], l)$
        $l \leftarrow l - f_{r-1}(i)$
        $\lambda \leftarrow$ EXTRACT($(B_r, (l-1 \bmod |\mathtt{exp}(B_r)|) + 1, \lambda)$)
        $k \leftarrow \lceil l / |\mathtt{exp}(B_r)| \rceil + 1$
        while $i \leq k_2 \land \lambda > 0$ do    // iterate on the subsequent blocks
            while $r \leq t \land \lambda > 0$ do    // iterate on the subsequent block symbols $B_r$
                while $k \leq i^{c_r} \land \lambda > 0$ do    // iterate within the copies of $B_r$
                    $\lambda \leftarrow$ EXTRACT($(B_r, 1, \lambda)$)
                    $k \leftarrow k + 1$
                $k \leftarrow 1$
                $r \leftarrow r + 1$
            $r \leftarrow 1$
            $i \leftarrow i + 1$
    return $\lambda$
```

Algorithm 11 Range minimum queries on SLPs of height h in $\mathcal{O}(h)$ time

```
Input   : A variable $A$ of an SLP and positions $1 \le p \le q \le |\texttt{exp}(A)|$.
Output: Returns $\texttt{rmq}(\texttt{exp}(A)[p\,..\,q])$ and the corresponding minimum value.
function $\textsc{rmq}(A, p, q)$
    if $(p, q) = (1, |\texttt{exp}(A)|)$ then return $\texttt{rmq}(A)$ (which is precomputed)
    else if $A \to BC$ then    // first see if we go only left or only right
        if $q \le |\texttt{exp}(B)|$ then return $\textsc{rmq}(B, p, q)$
        else if $p > |\texttt{exp}(B)|$ then return $\textsc{rmq}(C, p - |\texttt{exp}(B)|, q - |\texttt{exp}(B)|)$
        else $(p \le |\texttt{exp}(B)| < q)$  // else compose a left suffix and a right prefix call
            $\langle m_l, v_l \rangle \leftarrow \textsc{rmq}(B, p, |\texttt{exp}(B)|)$
            $\langle m_r, v_r \rangle \leftarrow \textsc{rmq}(C, 1, q - |\texttt{exp}(B)|)$
            if $v_l \le v_r$ then return $\langle m_l, v_l \rangle$
            else return $\langle |\texttt{exp}(B)| + m_r, v_r \rangle$
```

Algorithm 12 Next smaller values on SLPs of height h in $\mathcal{O}(h)$ time

```
Input   : A variable $A$ of an SLP, position $1 \le p \le |\texttt{exp}(A)|$, and threshold $v$.
Output: The position $\textsc{nsv}(\texttt{exp}(A), p, v)$.
function $\textsc{nsv}(A, p, v)$
    if $\texttt{rmq}(A).v \ge v$ then    // whole symbols detect failure immediately
        return $|\texttt{exp}(A)| + 1$
    else if $A \to a$ then
        return 1
    else if $A \to BC$ then
        if $p \le |\texttt{exp}(B)|$ then    // first try to find the answer inside $\texttt{exp}(B)$
            $p \leftarrow \textsc{nsv}(B, p, v)$
            if $p \le |\texttt{exp}(B)|$ then    // return the answer if found
                return $p$
        return $|\texttt{exp}(B)| + \textsc{nsv}(C, p - |\texttt{exp}(B)|, v)$    // else try on the whole $C$
```

Algorithm 13 Computation of general string functions in RLSLPs in $\mathcal{O}(\log n)$ steps

Input : A variable A of an RLSLP (with its arrays L and F as global variables), and two positions $1 \le i \le j \le |\mathtt{exp}(A)|$.

Output: $f(\mathtt{exp}(A)[i\,..\,j])$.

```
function F$(A, i, j)$
    if $(i, j) = (1, |\mathtt{exp}(A)|)$ then      // whole symbols solved in constant time
        return $F[A]$
    else if $A \to BC$ then      // try to recurse only left or right
        if $j \le |\mathtt{exp}(B)|$ then
            return F$(B, i, j)$
        else if $|\mathtt{exp}(B)| < i$ then
            return F$(C, i - |\mathtt{exp}(B)|, j - |\mathtt{exp}(B)|)$
        else $(i \le |\mathtt{exp}(B)| < j)$      // compose left suffix and right prefix calls
            $f_l \leftarrow$ F$(B, i, |\mathtt{exp}(B)|)$
            $f_r \leftarrow$ F$(C, 1, j - |\mathtt{exp}(B)|)$
            return $g(f_l, f_r)$
    else $(A \to B^t)$      // run-length rule spanning from $t'$ to $t''$
        $t' \leftarrow \lceil i/|\mathtt{exp}(B)| \rceil$
        $t'' \leftarrow \lceil j/|\mathtt{exp}(B)| \rceil$
        if $t' = t''$ then      // still recurse on only one symbol
            return F$(B, i - (t'-1) \cdot |\mathtt{exp}(B)|, j - (t'-1) \cdot |\mathtt{exp}(B)|)$
        $f_l \leftarrow$ F$(B, i - (t'-1) \cdot |\mathtt{exp}(B)|, |\mathtt{exp}(B)|)$      // left suffix call
        $f_r \leftarrow$ F$(B, 1, j - (t''-1) \cdot |\mathtt{exp}(B)|)$      // right prefix call
        Compute $f_c(t'' - t' - 1)$ using the recurrence      // many whole symbols

            $f_c(k) \leftarrow \begin{cases} f(\varepsilon) & \text{if } k = 0; \\ F[B] & \text{if } k = 1; \\ g(f_c(k/2), f_c(k/2)) & \text{if } k \text{ is even;} \\ g(F[B], f_c(k-1)) & \text{if } k \text{ is odd.} \end{cases}$

        return $g(g(f_l, f_c(t'' - t - 1)), f_r)$      // compose left, middle, right
```

Chapter 8

Extending Repetitiveness Measures to the Two-dimensional Space

Two-dimensional data, ranging from images to matrices, often contains inherent redundancy, wherein identical or similar substructures recur throughout the dataset. This great source of redundancy can be exploited for compression. Very recently, Brisaboa et al. [19] introduced the 2D Block Trees to compress images, graphs, and maps.

On the theoretical side, while in the one-dimensional case much attention has been given to the study and analysis of measures of repetitiveness to assess the performance of compressed indexing data structures [99, 100], in the two-dimensional context there is still no systematic study of measures that can effectively capture repetitiveness. In the one-dimensional case, an important role is played by the δ measure, which computes the maximum number of substrings of the same length that occur in a text, and by the γ measure, which represents the smallest sets of positions (string attractors) in the text at which all substrings can be found. These measures, although unreachable or unknown to be reachable, lower bound all other repetitiveness measures based on copy-paste mechanisms. Furthermore, the worst-case optimal space to represent a text can be expressed as a function of δ [79]. In the two-dimensional case, an important step in this direction has been made by Carfagna and Manzini in [22], where the repetitiveness measures δ and γ are extended to square two-dimensional strings, by exploring square substructures within the data. They have shown that some properties that hold for one-dimensional strings are still preserved in the two-dimensional case, and the space used by a two-dimensional block tree has been bounded in terms of their extension of δ.

In this chapter, we extend some repetitiveness measures to generic two-dimensional strings. The chapter is structured as follows.

- In Section 8.1, we introduce some general notions on 2D strings and give some examples.
- In Section 8.2, we generalize the measures δ and γ to the 2D setting, which differently from the measures defined by Carfagna and Manzini [22], use rectangular substrings, instead of squares, in their definition. We show that our measures, while retaining many of the properties valid in one-dimensional case, can behave very differently compared

to those defined by Carfagna and Manzini [22], even when applied to one-dimensional strings.

- In Section 8.3, we generalize straight-line programs (SLPs) and run-length straight-line programs (RLSLPs) to 2D strings. We introduce a new repetitiveness measure g based on 2D SLPs. We also introduce 2D RLSLPs and the correspondent repetitiveness measure $g_{\texttt{rl}}$.
- In Section 8.4, we introduce macro schemes for 2D strings that generalize bidirectional macro schemes. We also show that the mutual relationship among g, $g_{\texttt{rl}}$ and the size b of the smallest valid 2D macro scheme are the same as for one-dimensional strings.
- We show in Section 8.5, that the relationship between δ, γ, and the other repetitiveness measures are lost when they are extended to the 2D setting. Indeed, it is well known that for 1D strings the relationship $\delta \leq \gamma \leq b \leq g_{\texttt{rl}} \leq g$ holds. In the 2D setting, it can happen that $\delta = \Omega(g\sqrt[4]{N}/\log N)$ for some 2D string families, where N is the size of the 2D string.
- Finally, in Section 8.6, we use our generalized measures to analyze the effectiveness of *linearizations*, i.e., the transformation of the input matrix into a 1D string, which is then compressed. We measure the effectiveness of this technique for the simple row-by-row linearization, and show that the compression obtained can be substantially worse than what could be achieved using 2D SLPs.

Overall, our results shed some light on the difficulties of detecting and exploiting repetitiveness in the 2D setting, and show that some concepts/tools introduced in 1D are less effective in 2D.

We believe it could be worthwhile to explore 2D compression algorithms based on grammar compression (approaching the measures g and $g_{\texttt{rl}}$) and copy-paste (approaching the measure b) as methods to represent 2D strings.

8.1 Basics on 2D Strings

Let $\Sigma = \{a_1, a_2, \ldots, a_\sigma\}$ be an ordered alphabet. A *2D string* $M_{m\times n}$ is a $(m \times n)$-matrix with $m \geq 1$ *rows* and $n \geq 1$ *columns* such that each element $M[i][j]$ belongs to Σ. The size of $M_{m\times n}$ is $N = mn$. Note that a position in $M_{m\times n}$ consists of a pair (i, j), with $1 \leq i \leq m$ and $1 \leq j \leq n$. Note that traditional one-dimensional strings are a special case of 2D strings with $m = 1$. We denote by $\Sigma^{m\times n}$ the set of all 2D strings with m rows and n columns over Σ. A 2D string in $\Sigma^{m\times n}$ is called a *square* if $m = n$.

The concatenation between two matrices is a partial operation that can be performed horizontally ($\varobar$) or vertically ($\ominus$), with the constraint that the number of rows or columns coincide, respectively. Such operations have been described in [57] where concepts and techniques of formal languages have been generalized to two dimensions. We illustrate how 2D strings can be concatenated in Example 8.1.1.

Example 8.1.1 Consider the 2D strings

$$A = \begin{bmatrix} \mathtt{a} & \mathtt{b} & \mathtt{a} \\ \mathtt{a} & \mathtt{b} & \mathtt{b} \end{bmatrix}, B = \begin{bmatrix} \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{b} \\ \mathtt{b} & \mathtt{b} & \mathtt{b} & \mathtt{b} \end{bmatrix}, \text{ and } C = \begin{bmatrix} \mathtt{a} & \mathtt{a} & \mathtt{a} \\ \mathtt{a} & \mathtt{a} & \mathtt{b} \\ \mathtt{a} & \mathtt{b} & \mathtt{b} \end{bmatrix}.$$

We can obtain new 2D strings by using $\oplus$ and $\ominus$, respectively:

$$A \oplus B = \begin{bmatrix} \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{b} \\ \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{b} & \mathtt{b} & \mathtt{b} & \mathtt{b} \end{bmatrix}, \text{ and } A \ominus C = \begin{bmatrix} \mathtt{a} & \mathtt{b} & \mathtt{a} \\ \mathtt{a} & \mathtt{b} & \mathtt{b} \\ \mathtt{a} & \mathtt{a} & \mathtt{a} \\ \mathtt{a} & \mathtt{a} & \mathtt{b} \\ \mathtt{a} & \mathtt{b} & \mathtt{b} \end{bmatrix}.$$

Note that $A \ominus B$ and $A \oplus C$ are undefined.

We denote by $M_{m\times n}[i_1 \mathinner{.\,.} i_2][j_1 \mathinner{.\,.} j_2]$ the submatrix starting at position (i_1, j_1) and ending at position (i_2, j_2). We say that a 2D string F is a *factor* or *substring* of $M_{m\times n}$ if there exist two positions (i_1, j_1) and (i_2, j_2) such that $F = M_{m\times n}[i_1 \mathinner{.\,.} i_2][j_1 \mathinner{.\,.} j_2]$.

Definition 8.1.2 Given a 2D string $M_{m\times n}$, the *2D substring complexity* function P_M counts for each pair of positive integers (k_1, k_2) the number of distinct $(k_1 \times k_2)$-factors in $M_{m\times n}$.

We show how P_M works in Example 8.1.3.

Example 8.1.3 Consider the 2D strings

$$M = \begin{bmatrix} \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{b} \\ \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{b} \\ \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{b} \\ \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{b} \\ \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{b} \end{bmatrix}, F_1 = \begin{bmatrix} \mathtt{a} & \mathtt{b} \\ \mathtt{a} & \mathtt{b} \\ \mathtt{a} & \mathtt{b} \end{bmatrix}, F_2 = \begin{bmatrix} \mathtt{a} & \mathtt{a} \\ \mathtt{a} & \mathtt{a} \end{bmatrix}, F_3 = \begin{bmatrix} \mathtt{b} & \mathtt{b} \\ \mathtt{b} & \mathtt{b} \end{bmatrix} \text{ and } F_4 = \begin{bmatrix} \mathtt{a} & \mathtt{b} \\ \mathtt{a} & \mathtt{b} \end{bmatrix}.$$

The 2D string F_1 is a (3×2)-factor of M, as $F_1 = M[2 \mathinner{.\,.} 4][2 \mathinner{.\,.} 3]$. Moreover, it can be verified that F_2, F_3 and F_4 are the only (2×2)-factors of M. Hence, $P_M(2, 2) = 3$.

8.2 Measures δ and γ in two dimensions

We start by extending the combinatorial measures δ and γ. The following definition extends the measure δ to 2D strings.

Definition 8.2.1 Let $M_{m\times n}$ be a 2D string and P_M be the 2D substring complexity function of $M_{m\times n}$. Then, $\delta(M_{m\times n}) = \max\{P_M(k_1, k_2)/(k_1k_2), 1 \le k_1 \le m, 1 \le k_2 \le n\}$.

Note that for 1D strings (i.e., when $m = 1$) the above definition coincides with the one-dimensional version of δ. Recently, Carfagna and Manzini introduced another alternative extension of δ, here denoted by $\delta_\square$ [22], which is limited to square 2D input strings and only considers *square* factors for computing the substring complexity. Below we report the definition of such a measure, applied to a generic two-dimensional string.

Definition 8.2.2 Let $M_{m\times n}$ be a 2D string and P_M be the 2D substring complexity of $M_{m\times n}$. Then, $\delta_\square(M_{m\times n}) = \max\{P_M(k,k)/k^2, 1 \le k \le \min\{m,n\}\}$.

From the definitions of $\delta_\square$ and δ, the following lemma easily follows.

Lemma 8.2.3 For every 2D string $M_{m\times n}$ it holds that $\delta(M_{m\times n}) \ge \delta_\square(M_{m\times n})$.

Although the two measures δ and $\delta_\square$ may seem similar, considering square factors instead of rectangular ones may result in very different values. Example 8.2.4 shows how different the two measures can be when applied to one-dimensional strings, while Example 8.2.5 shows that there exist families of square 2D strings for which $\delta_\square = o(\delta)$.

Example 8.2.4 Given a 1D string $S \in \Sigma^n$, let $M_{1\times n} \in \Sigma^{1\times n}$ be the matrix such that $M_{1\times n}[1][1\,..\,n] = S[1\,..\,n]$. Since the only squares that occur in $M_{1\times n}$ are the factors of size 1×1, it holds $\delta_\square(M_{1\times n}) = P_M(1,1)/1^2 \le |\Sigma|$. On the other hand, $\delta(M_{1\times n}) = \delta(S)$.

Example 8.2.5 Let $M_{n\times n}$ be the square 2D string in [22, Lemma 4]. Assuming n is a perfect square, the first row of $M_{n\times n}$ is the string $S = B_1B_2\,..\,B_{\sqrt{n}/2}$ composed by $\sqrt{n}/2$ blocks, each one of size $2\sqrt{n}$, with $B_i = \mathtt{1}^i\mathtt{0}^{(2\sqrt{n}-i)}$. The remaining rows of $M_{n\times n}$ are all $\mathtt{\#}^n$. In [22, Lemma 4] it is shown that $\delta_\square(M_{n\times n}) = \mathcal{O}(1)$. On the other hand, note that for $i \in [2\,..\,\sqrt{n}/2]$ and $j \in [0\,..\,\sqrt{n}-i]$, the strings $\mathtt{0}^j\mathtt{1}^i\mathtt{0}^{\sqrt{n}-j-i}$ are all different substrings of length $\sqrt{n}$ of S. Since these substrings are in total $\Omega(n)$, it holds $\delta(M_{n\times n}) = \Omega(\sqrt{n})$.

The following definition generalizes to 2D strings the notion of string attractor.

Definition 8.2.6 A *2D string attractor* for a 2D string $M_{m\times n}$ is a set $\Gamma \subseteq [1\,..\,m] \times [1\,..\,n]$ with the property that any substring $M[i\,..\,j][k\,..\,l]$ of $M_{m\times n}$ has an occurrence $M[i'\,..\,j'][k'\,..\,l']$ such that $\exists(x,y) \in \Gamma$ with $i' \le x \le j'$ and $k' \le y \le l'$. The size of the smallest attractor for $M_{m\times n}$ is denoted by $\gamma(M_{m\times n})$.

When $m = 1$ the above definition coincides with the one for 1D strings, hence the measure γ inherits the properties for the one-dimensional case. In particular: γ is not monotone and computing $\gamma(M_{m\times n})$ is NP-hard [93, 71]. In addition, the following property holds.

Proposition 8.2.7 For every 2D string $M_{m\times n}$, it is $\delta(M_{m\times n}) \le \gamma(M_{m\times n})$.

Proof. Just as in the 1D case, for any 2D string M it holds $P_M(k_1,k_2) \le k_1k_2\gamma(M)$. □

The next proposition shows that in the 2D context, the gap between δ and γ can be larger than the one-dimensional case, where it is logarithmic [79].

Proposition 8.2.8 For every $m,n \ge 1$ there exists a 2D string $M_{m\times n}$ such that $\delta(M_{m\times n}) = \mathcal{O}(1)$ and $\gamma(M_{m\times n}) = \Omega(\min(m,n))$.

Proof. Let I_k be the $k \times k$ identity matrix. For every $m,n \ge 1$, let us consider the 2D string $M_{m\times n}$ such that $M_{m\times n}[1\,..\,\min(m,n)][1\,..\,\min(m,n)] = I_{\min(m,n)}$, and all the remaining

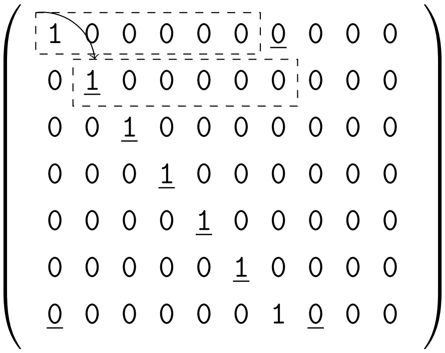


Figure 8.1: 2D string attractor for the matrix $M_{m\times n}$ of Proposition 8.2.8. The cells whose positions belong to the string attractor are underlined. We show how $M[1\,..\,1][1\,..\,6]$ has an occurrence $M[2\,..\,2][1\,..\,6]$ crossing the string attractor position $(2,2)$.

symbols are `0`'s. When either $m = 1$ or $n = 1$ the proof is trivial from known results on 1D strings, so let us assume $m, n \geq 2$. Let us further assume $m < n$. Next we show that $\Gamma = \{(2,2)\,..\,(m-1,m-1)\} \cup \{(1,m),(m,1),(m,m+1)\}$ is an attractor for $M_{m\times n}$. All the substrings of $M_{m\times n}$ that contain at least two occurrences of `1`'s have an occurrence crossing the position (i,i), for some $1 < i < m$, while all the substrings that consist of only `0`'s have an occurrence aligned with one of the `0`'s at position $(1,m)$, $(m,1)$, or $(m,m+1)$. The remaining factors contain only one occurrence of `1`'s that do not cross any position in Γ. These factors of size $k_1 \times k_2$ have to cross either the `1` in position $(1,1)$ or in position (m,m), and therefore it is either $k_1 = 1$ and $k_2 < m$, or vice versa. Observe that all these factors have another occurrence either starting in $(2,2)$ or ending in $(m-1,m-1)$, and therefore Γ is an attractor of $M_{m\times n}$. Since $M_{m\times n}$ has $m+1$ distinct columns the above attractor has minimum size, i.e., $\gamma(M_{m\times n}) = m+1$. We illustrate these string attractors in Figure 8.1. On the other hand, there exist at most $k_1 + k_2$ distinct substrings of size $k_1 \times k_2$ in $M_{m\times n}$: $k_1 + k_2 - 1$ correspond to substrings where the diagonal of $M_{m\times n}$ touches ones of the positions in the left or upper borders of the factor; the last one is the string of only `0`'s. Hence $\delta(M_{m\times n}) \leq 2$. The case $m > n$ is treated symmetrically by considering the attractor $\Gamma' = \{(2,2)\,..\,(n-1,n-1)\} \cup \{(1,n),(n,1),(n+1,n)\}$. For the case $n = m$ it is $M_{m\times n} = I_n$ and reasoning as above it is easy to see that $\Gamma'' = \{(2,2)\,..\,(n-1,n-1)\} \cup \{(1,n),(n,1)\}$ of size n is a minimal attractor for I_n and that $\delta(I_n) \leq 2$. □

In [22] the authors introduced an alternative definition of string attractors for square 2D input strings in which they consider only square factors. We can define such a measure, denoted by $\gamma_\square$, also for generic 2D strings, by simply considering only square substrings of $M_{m\times n}$ in Definition 8.2.6. From the definitions of γ and $\gamma_\square$ we immediately get the following relationship:

Lemma 8.2.9 For every 2D string $M_{m\times n}$ it holds that $\gamma(M_{m\times n}) \geq \gamma_\square(M_{m\times n})$.

The following example shows that γ and $\gamma_\square$ can be asymptotically different.

Example 8.2.10 Consider again the $m \times m$ identity matrix I_m. For each $k \leq m$, a $k \times k$ square factor of I_m either consists of i) all `0`'s, or ii) all `0`'s except only one diagonal composed by $\mathtt{1}$'s. Hence, all square factors of type i) have an occurrence that includes position $(m, 1)$ (i.e. the bottom left corner), while all those of type ii) have an occurrence that includes the position $(\lfloor m/2 \rfloor, \lfloor m/2 \rfloor)$ (i.e. the $\mathtt{1}$ at the center). It follows that $\gamma_{\Box}(I_m) = 2 \in \mathcal{O}(1)$, while from the proof of Proposition 8.2.8 it can be deduced that $\gamma(I_m) = \Theta(m)$.

The measures δ and γ inherit from the 1D case the property that δ is unreachable and γ is unknown to be reachable [99]. In the next sections, we show how to generalize to the 2D case the measures g, $g_{\mathtt{rl}}$, and b which are reachable both in 1D and 2D.

8.3 (Run-length) Straight-line Programs for 2D Strings

In this section we consider a generalization of straight-line programs for the two-dimensional space, which was first introduced in [11], and we use it to generalize the measures g and $g_{\mathtt{rl}}$ to 2D strings.

Definition 8.3.1 Let $M_{m \times n}$ be a 2D string. A *2-dimensional straight-line program* (2D SLP) for $M_{m \times n}$ is a 2D context-free grammar $G = (V, \Sigma, R, S)$ where V is the set of variables, Σ is the set of terminals disjoint from V, R is the set of rules (there is exactly one rule per variable), and S is the initial variable; such that it uniquely generates $M_{m \times n}$. The definition of the right-hand side of a variable can have the form

$$A \to a,\ A \to B \varobar C,\ \text{or } A \to B \ominus C,$$

where $a \in \Sigma$ and $B, C \in V$. We call these definitions *terminal rules*, *horizontal rules*, and *vertical rules*, respectively. The expansion of a variable is defined as

$$\mathtt{exp}(A) = a,\ \mathtt{exp}(A) = \mathtt{exp}(B) \varobar \mathtt{exp}(C),\ \text{ or } \mathtt{exp}(A) = \mathtt{exp}(B) \ominus \mathtt{exp}(C),$$

respectively. The size of a 2D SLP is the sum of the sizes of all right-hand sides of the rules of the grammar, where we assume the terminal rules have size 1, and the horizontal and vertical rules have size 2.

Definition 8.3.2 The measure $g(M_{m \times n})$ is defined as the size of the smallest 2D SLP generating $M_{m \times n}$.

It is easy to see that the above definition coincides with that of the measure g for one-dimensional strings if only horizontal concatenations are considered.

Similarly to regular SLPs, 2D SLPs cannot compress to constant space.

Proposition 8.3.3 It always holds that $g(M_{m \times n}) = \Omega(\log(mn))$.

Proof. From the initial variable S, each substitution step can double the size of the current 2D string of variables. Hence, a 2D SLP of size g can produce a 2D string of size at most $2^{\lfloor g/2 \rfloor}$. Therefore, a string of size $N = mn$ needs a grammar of size $\Omega(\log_2(mn))$. $\square$

Proposition 8.3.4 The problem of determining if there exists a 2D SLP of size at most k generating a text $M_{m\times n}$ is NP-complete.

Proof. Observe that the 1D version of the problem, known to be NP-complete [31], reduces to the 2D version by considering 1D strings as matrices of size $1 \times n$. □

As in the 1D case, we can extend 2D SLPs with *run-length rules*, obtaining more powerful grammars.

Definition 8.3.5 A *2-dimensional run-length straight-line program* (2D RLSLP) is a 2D SLP that in addition allows special rules, which are assumed to be of size 2, of the form

$$A \rightarrow \obar^{k} B \text{ and } A \rightarrow \ominus^{k} B$$

for $k > 1$, with their expansions defined as

$$\mathtt{exp}(A) = \underbrace{\mathtt{exp}(B) \obar \mathtt{exp}(B) \obar \cdots \obar \mathtt{exp}(B)}_{k \text{ times}}$$
$$\mathtt{exp}(A) = \underbrace{\mathtt{exp}(B) \ominus \mathtt{exp}(B) \ominus \cdots \ominus \mathtt{exp}(B)}_{k \text{ times}}$$

respectively.

Definition 8.3.6 The measure $g_{\mathtt{rl}}(M_{m\times n})$ is defined as the size of the smallest 2D RLSLP generating $M_{m\times n}$.

Proposition 8.3.7 For every 2D string $M_{m\times n}$ it holds that $g_{\mathtt{rl}}(M_{n\times n}) \leq g(M_{m\times n})$. Moreover, there are infinite string families where $g_{\mathtt{rl}} = o(g)$.

Proof. The first claim is trivial by definition. The second claim is proven by considering the family of $1 \times n$ matrices $M_{1\times n} = \mathtt{a}^n$ for which $g = \Omega(\log n)$ and $g_{\mathtt{rl}} = \mathcal{O}(1)$. □

A useful definition for 2D SLPs/RLSLPs is the following.

Definition 8.3.8 The *grammar tree* of a 2D SLP is an ordered labeled tree where S is the root, and the children of a variable A are the variables in its right-hand side. The tree is pruned, so only the first occurrence of each variable (i.e., the leftmost occurrence at the highest height) is an internal node, and the remaining occurrences are leaves. For 2D RLSLPs, each horizontal run-length variable has two children: B, which is a leaf only if it is the first occurrence of B, and $\obar^{k-1}B$, which is always a leaf. The treatment is analogous for vertical rules.

The size of the grammar tree is proportional to the size of the 2D SLP/RLSLP, as each variable appears only once as internal node.

8.4 Macro Schemes for 2D Strings

The notion of macro scheme and the corresponding measure b can be naturally generalized to 2D strings with the following definition.

Definition 8.4.1 A *2D macro scheme* for a string $M_{m\times n}$ is any factorization of $M_{m\times n}$ into a set of disjoint phrases (i.e., a multiset of substrings of $M_{m\times n}$ that can be concatenated together to form $M_{m\times n}$) such that any phrase is either a square of dimension 1×1 called an *explicit symbol/phrase*, or it is a copied phrase with rectangular shape whose source in $M_{m\times n}$ starts at a different position. For a 2D macro scheme to be *valid* or *decodable*, there must exist a function $\mathtt{map} : ([1\,..\,m] \times [1\,..\,n]) \cup \{\bot\} \to ([1\,..\,m] \times [1\,..\,n]) \cup \{\bot\}$ such that:

1. $\mathtt{map}(\bot) = \bot$, and if $M[i][j]$ is an explicit symbol, then $\mathtt{map}(i, j) = \bot$;
2. for each copied phrase $M[i_1\,..\,j_1][i_2\,..\,j_2]$, it must hold that $\mathtt{map}(i_1 + t_1, i_2 + t_2) = \mathtt{map}(i_1, i_2) + (t_1, t_2)$ for $(t_1, t_2) \in [0\,..\,j_1 - i_1] \times [0\,..\,j_2 - i_2]$, where $\mathtt{map}(i_1, i_2)$ is the upper left corner of the source for $M[i_1\,..\,j_1][i_2\,..\,j_2]$;
3. for each $(i, j) \in [1\,..\,m] \times [1\,..\,n]$ there exists $k > 0$ such that $\mathtt{map}^k(i, j) = \bot$.

Definition 8.4.2 We define $b(M_{m\times n})$ as the size of the smallest valid 2D macro scheme for $M_{m\times n}$.

We illustrate a macro scheme for the 7×7 identity matrix I_7 in Figure 8.2. We carefully describe the underlying map of this macro scheme in Example 8.4.3.

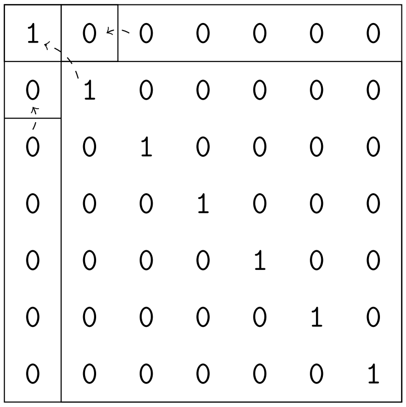
Example 8.4.3 Let I_n be the $n \times n$ identity matrix. A macro scheme for I_n consists of the phrases $\{X_1, X_2, X_3, X_4, X_5, X_6\}$ where: i) $X_1 = I_n[1][1]$ is an explicit symbol (the $\mathtt{1}$ in the top-left corner); ii) $X_2 = I_n[1][2]$ is an explicit symbol; $X_3 = I_n[2][1]$ is an explicit symbol; $X_4 = I_n[1][3\,..\,n]$ is a phrase with source $(1, 2)$; $X_5 = I_n[3\,..\,n][1]$ is a phrase with source $(2, 1)$; and $X_6 = I_n[2\,..\,n][2\,..\,n]$ is a phrase with source $(1, 1)$. The underlying function $\mathtt{map}$ is defined as $\mathtt{map}(1, 1) = \mathtt{map}(1, 2) = \mathtt{map}(2, 1) = \bot$, $\mathtt{map}(1, j) = (1, j - 1)$ for $j \in [3\,..\,n]$, $\mathtt{map}(i, 1) = (i - 1, 1)$ for $i \in [3\,..\,n]$, and $\mathtt{map}(i, j) = (i - 1, j - 1)$ for $i, j \in [2\,..\,n] \times [2\,..\,n]$. One can see that $\mathtt{map}^n(i, j) = \bot$ for each i and j. Hence, the macro scheme is valid and $b(I_n) \le 6$.

The following two propositions show that the computability properties of b and its relationship with the measures $g_{\mathtt{rl}}$ and g are preserved in the 2D context.

Proposition 8.4.4 The problem of determining if there exists a valid 2D macro scheme of size at most k for a text $M_{m\times n}$ is NP-complete.

Proof. The 1D version of the problem, which is known to be NP-complete [51], reduces to the 2D version of the problem in constant time. □

Proposition 8.4.5 For every 2D string $M_{m\times n}$ it holds that $b(M_{m\times n}) \le g_{\mathtt{rl}}(M_{m\times n})$. Moreover, there are string families where $b = o(g_{\mathtt{rl}})$.



1	0	0	0	0	0	0
0	1	0	0	0	0	0
0	0	1	0	0	0	0
0	0	0	1	0	0	0
0	0	0	0	1	0	0
0	0	0	0	0	1	0
0	0	0	0	0	0	1

Figure 8.2: Macro scheme with 6 phrases for I_7. The entries $(1,1)$, $(1,2)$, and $(2,1)$ are explicit symbols. The remaining phrases point to the source from where they are copied.

Proof. We show how to construct a macro scheme from a 2D RLSLP, representing the same 2D string and having the same asymptotic size. Let G be a 2D RLSLP generating $M_{m\times n}$ and consider its grammar tree.

Each leaf of the grammar tree corresponding to a variable that expands to a single symbol at cell $M[i][j]$, induces an explicit phrase of the parsing at that specific cell.

A leaf of the grammar tree corresponding to an occurrence of the variable A expanding at cells $M[i_1\,..\,i_2][j_1\,..\,j_2]$ becomes a phrase of the parsing at these cells, and its source is aligned with the upper left corner of the expansion on $M_{m\times n}$ of the internal occurrence of A in the grammar tree.

For a leaf $\obar^{k-1}B$ induced by an horizontal run-length variable, which expands to $M[i_1\,..\,i_2][j_1\,..\,j_2]$ in $M_{m\times n}$, we construct a phrase at these cells, pointing to the occurrence of $\texttt{exp}(B)$ at its left in $M_{m\times n}$. We do analogously, for 2D vertical run-length rules.

This parsing is decodable and its size is bounded by the size $g_{\texttt{rl}}$ of the grammar. For a family where $b = o(g_{\texttt{rl}})$, this holds for the 1D version [102]. □

8.5 Differences Between the 1D and the 2D Setting

In the 1-dimensional context, for each string $S \in \Sigma^*$ it holds that $\delta \leq \gamma \leq b \leq g_{\texttt{rl}} \leq g$. In particular, while $\delta \leq \gamma$ and $b \leq g_{\texttt{rl}} \leq g$ truly rely on their definitions, the missing link between γ and b has been proved by Kempa and Prezza by showing how any macro scheme of size b induces a suitable string attractor with at most $2b$ positions [71]. Later, Bannai et al. [6] showed that for 1D strings there is a separation between γ and b by using the family of Thue-Morse words, a string family for which $\gamma = \mathcal{O}(1)$ and $b = \Theta(\log n)$.

In the previous sections, we have shown that on the 2D setting the same relationships between δ and γ hold, as well as the one between b, $g_{\texttt{rl}}$, and g. However, unlike the 1D setting, we can have 2D strings for which the measure b is asymptotically smaller than γ.

Proposition 8.5.1 There exists a 2D string family where $\gamma = \Omega(b\sqrt{N})$, where N is the size of the 2D string.

Proof. As shown in the proof of Proposition 8.2.8, $\gamma = \Omega(n)$ in the family of identity matrices I_n. On the other hand, in Example 8.4.3 we showed how to construct a macro scheme with only 6 phrases for the same family of strings, i.e., $b = \mathcal{O}(1)$. The claim follows since for every identity matrix I_n it holds that $n = \sqrt{N}$. □

It follows that, when considering a 2D setting, the measure b can be much better than γ. Hence, both measures are uncomparable with each other. A follow up question is whether the relationship between the measures δ and b on the 2-dimensional context is preserved. As the following proposition shows, not only the measure b can be asymptotically smaller than δ, but it can be asymptotically smaller than $\delta_{\square}$ too.

Proposition 8.5.2 There exists a 2D string family where $\delta_{\square} = \Omega(b\sqrt[4]{N})$, where N is the size of the 2D string.

Proof. Let $k > 3$. Let $\mathcal{F}(k)$ be a set containing all the 2D strings of dimensions $k \times k$ where:

1. exactly two cells, in distinct rows, contain a `1`;
2. the `1` on the row above cannot be more to the right than the `1` on the row below;
3. all the remaining cells contain `0`'s.

There are $\binom{k}{2}$ ways to choose two distinct rows to verify condition 1), and then k^2 ways to choose which cells in these rows contain the `1`'s. However, only $k(k+1)/2$ of these 2D strings satisfy condition 2). Hence, there exist exactly $k^2(k-1)(k+1)/4$ such strings in $\mathcal{F}(k)$.

Let us now construct a 2D string A_k containing all the $k \times k$ strings in $\mathcal{F}(k)$ as substrings. Let $B(i,j)$ be the $k \times k$ string containing a `1` in position $(1,1)$, and another `1` in position (i,j). The matrix A_k is defined as follows: for each $i \in [2\,..\,k]$ take the $k \times k^2$ substring containing only `0`'s and append it below the $k \times k^2$ matrix $B(i,1) \obar B(i,2) \obar \cdots \obar B(i,k)$. Then, concatenate all these matrices from top to bottom. Finally, append to the left and to the right a $(k-1)2k \times k$ substring containing only `0`'s. Schematically, the matrix A_k has the following form:

$$\begin{array}{lllll}
\mathtt{0}_{k\times k} & \mathtt{0}_{k\times k} & \dots & \mathtt{0}_{k\times k} & \mathtt{0}_{k\times k} \\
\mathtt{0}_{k\times k} & B(2,1) & \dots & B(2,k) & \mathtt{0}_{k\times k} \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
\mathtt{0}_{k\times k} & \mathtt{0}_{k\times k} & \dots & \mathtt{0}_{k\times k} & \mathtt{0}_{k\times k} \\
\mathtt{0}_{k\times k} & B(k,1) & \dots & B(k,k) & \mathtt{0}_{k\times k}
\end{array}$$

We can move a $k \times k$ window containing both `1`'s of some matrix $B(i,j)$ for some i and j to find any string in $\mathcal{F}(k)$ as a substring of A_k. Thus, it holds that $\delta_{\square}(A_k) = \Omega(k^2)$. Let $i \in [2\,..\,k]$ and $j \in [1\,..\,k]$. Moreover, the size of A_k is $N = m \times n = 2k(k-1) \times k(k+2) = \Theta(k^4)$.

Now we show how to construct a valid macro scheme for A_k. The intuition for the macro scheme is to first create phrases for the rectangles containing only 0's surrounding the central part of A_k. Then, we observe that the submatrices $B(i,1) \oplus B(i,2) \oplus \cdots \oplus B(i,k)$ contain only 3 types of rows: 1) $\mathtt{0}^{k^2}$; 2) $(\mathtt{10}^{k-1})^k$; or 3) $(\mathtt{10}^k)^{k-1}\mathtt{1}$. Thus, we can use $\mathcal{O}(1)$ phrases for the first occurrence (at the top) of rows of type 2) and 3), and then use them as a reference for the other occurrences. More in detail, the macro scheme contains the phrases $\{X_1, \ldots, X_{12}\}$ where: i) $X_1 = A_k[1][1]$ is an explicit phrase containing a 0; ii) $X_2 = A_k[2\mathinner{.\,.}k][1]$ is a phrase with source $(1,1)$; iii) $X_3 = A_k[1\mathinner{.\,.}k][2\mathinner{.\,.}n]$ is a phrase with source $(1,1)$; iv) $X_4 = A_k[k{+}1\mathinner{.\,.}m][1\mathinner{.\,.}k]$ is a phrase with source $(1,1)$; v) $X_5 = A_k[k{+}1\mathinner{.\,.}m][k(k{+}1){+}1\mathinner{.\,.}n]$ is a phrase with source $(1,1)$; vi) $X_6 = A_k[k+1][k+1]$ is an explicit phrase containing a 1; vii) $X_7 = A_k[k+1][k+2\mathinner{.\,.}2k]$ is a phrase with source $(1,1)$; viii) $X_8 = A_k[k+1][2k+1\mathinner{.\,.}k(k+1)]$ is a phrase with source $(k+1,k+1)$; ix) $X_9 = A_k[k+2][k+1]$ is an explicit phrase containing a 1; x) $X_{10} = A_k[k{+}2][k{+}2\mathinner{.\,.}2k{+}1]$ is a phrase with source $(1,1)$; xi) $X_{11} = A_k[k{+}2][2k{+}2\mathinner{.\,.}k(k{+}1)]$ is a phrase with source $(k+2,k+1)$; xii) $X_{12} = A_k[k+3\mathinner{.\,.}2k][k+1\mathinner{.\,.}k(k+1)]$ is a phrase with source $(1,1)$.

Observe that the remaining phrases refer to the matrix $A_k[2k+1\mathinner{.\,.}m][k+1\mathinner{.\,.}k(k+1)]$, with rows of the type 1), 2), and 3) described above. Hence, each range of consecutive rows of type 1) (i.e. of all 0's) can be copied from the (biggest) block of consecutive rows of 0's starting at the beginning of phrase X_{12}, and we have $2(k-2)$ of such phrases. For each row of type 2) we use a single phrase pointing to the beginning of its first occurrence, at the beginning of phrase X_6, and we have exactly $k-2$ of such rows. Analogously, for each of the $k-2$ rows of type 3), we use a single phrase pointing to the beginning of phrase X_9. Thus, the total size of the macro scheme built is indeed $12 + 4(k-2) = \mathcal{O}(k)$. Hence, we proved that $\delta_\square = \Omega(bk)$. As $k = \Theta(\sqrt[4]{N})$, the claim follows. $\square$

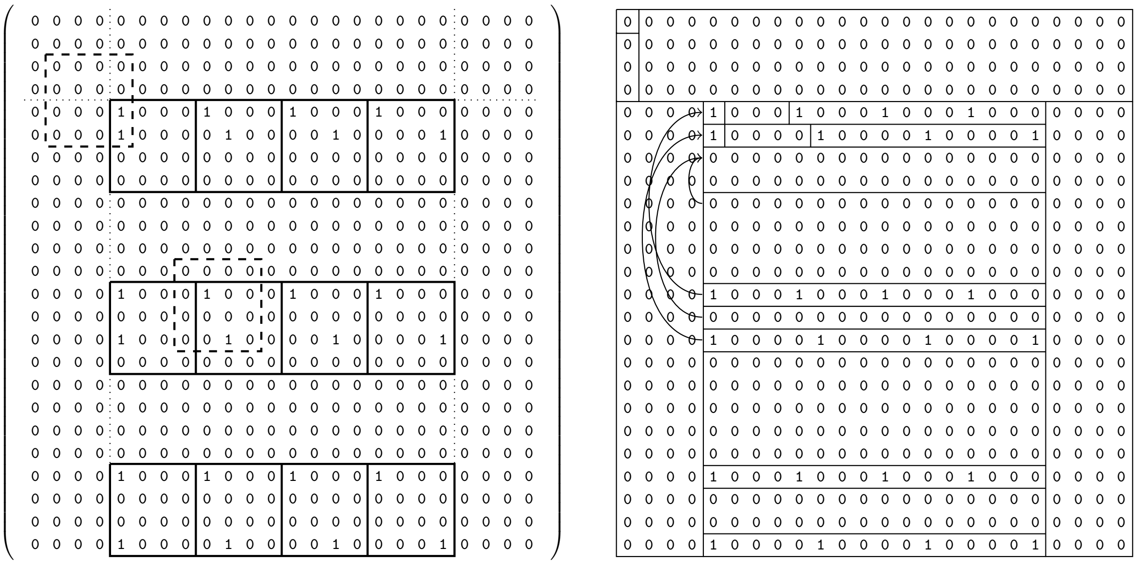


Figure 8.3: Matrix A_k for $k = 4$. To the left, we highlight the substrings $B(i,j)$, and show how to obtain other strings in $\mathcal{F}(k)$ by moving a 4×4 window. To the right, we show the macro scheme described in Proposition 8.5.2, which is formed by exactly $4(k+1)$ phrases. Each arrow shows the source from where the phrases are copied.

An example on A_4 can be seen in Figure 8.3. Observe that we can obtain the same result if we compare b and $\delta_\square$ on a square matrix, i.e., with the same setting from [22]. In fact, since the measure $\delta_\square$ is monotone, we can append columns of $\mathtt{0}$'s to the left or to the right of A_k until we obtain a square preserving the lower bound $\delta_\square = \Omega(k^2)$, while a suitable macro scheme may require at most $\mathcal{O}(1)$ new phrases.

As $\delta_\square \leq \delta$ for all 2D strings, we derive the following corollary.

Corollary 8.5.3 There exists a 2D string family where $b = o(\delta)$.

Even 2D SLPs can be noticeably smaller than $\delta_\square$.

Proposition 8.5.4 There exists a 2D string family where $\delta = \Omega(g\sqrt[4]{N}/\log N)$, where N is the size of the 2D string.

Proof. We show that in the same family of strings A_k of Proposition 8.5.2, it holds that $g = \mathcal{O}(\sqrt[4]{N}\log N)$ whereas $\delta_\square = \Omega(\sqrt{N})$. Notice that it is always possible to generate a string $\mathtt{0}_{k_1 \times k_2}$ with a 2D SLP of size $\Theta(\log(k_1 k_2))$. Hence, we can obtain 2D SLPs generating all the phrases of this type in the macro scheme of Proposition 8.5.2, having total size bounded by $\mathcal{O}(\log N^b) = \mathcal{O}(b\log N) = \mathcal{O}(\sqrt[4]{N}\log N)$. Similarly, to generate $(\mathtt{10}^{k-1})^k$ and $(\mathtt{10}^k)^{k-1}\mathtt{1}$ we need 2D SLPs whose size sums to $\mathcal{O}(\log k) = \mathcal{O}(\log N)$. Finally, we need $\mathcal{O}(\sqrt[4]{N}\log N)$ new rules to merge all the 2D SLPs described before. The total size of the 2D SLP is indeed $\mathcal{O}(\sqrt[4]{N}\log N)$. Thus, the result follows. □

Corollary 8.5.5 There exists a string family where $g = o(\delta)$.

We argue that the uncomparability of δ (or $\delta_\square$) and γ (or $\gamma_\square$) with g is enough to conclude that δ and γ are weak measures when it comes to consider 2-dimensional strings. Some improved versions of these results can be found in the conference version of this chapter [24].

8.6 Effectiveness of Linearization Techniques

A classical heuristic for compressing 2D strings is to transform a matrix $M_{m\times n}$ into a 1D string S and use a one-dimensional compressor on S. Having generalized 1D measures to 2D strings, it is natural to measure the effectiveness of linearization techniques by comparing, for a given measure μ, the values $\mu(M_{m\times n})$ and $\mu(S)$ where S is a linearization of $M_{m\times n}$. Clearly, for each matrix, there exists a linearization that makes the 2D string highly compressible: we can visit in order from left to right and from top to bottom all the occurrences of $a_1 \in \Sigma$, followed by all the occurrences of $a_2 \in \Sigma$, and so on, obtaining a string consisting in $|\Sigma|$ equal-letter runs. However, this method requires an ad-hoc linearization for each matrix which may require substantial additional information to retrieve the original input. It is therefore customary in the literature to consider linearization techniques that can be inverted efficiently in terms of both time and space.

One of the simplest linearization techniques consists in mapping $M_{m\times n}$ to the string $\mathtt{rlin}(M_{m\times n}) = \bigodot_{i=1}^{i=m} M[i][1..n] = M[1][1..n]\cdots M[m][1..n]$, obtained by concatenating its rows. The (lack of) effectiveness of this simple technique with respect to grammar compression has been already shown [11, Theorem 2] with an example of a matrix T_n of size $(2^{n+1}-1)\times(2^n+1)2^n$ such that $g(T_n)=\mathcal{O}(n)$, while $g(\mathtt{rlin}(T_n))=\Omega(2^n)$. The following proposition shows a similar result for the measure δ.

Proposition 8.6.1 There exists a family of square matrices $M_{n\times n}$ such that $\delta(M_{n\times n})=\mathcal{O}(1)$ and $\delta(\mathtt{rlin}(M_{n\times n}))=\Omega(n)$.

Proof. Let $M_{n\times n}$ be obtained by appending to the identity matrix I_{n-1} a row of $\mathtt{0}$'s at the bottom, and then a column of $\mathtt{1}$'s at the right. For each k_1, k_2, $P_M(k_1,k_2)$ is at most $3(k_1+k_2)$. We can see this by considering three cases: the submatrices that do not intersect the last row or column, the submatrices intersecting the last row, and the submatrices intersecting the last column. In each case, the distinct submatrices are associated to where the diagonal of $\mathtt{1}$'s intersects a submatrix (if it does so). This can happen in at most k_1+k_2 different ways. As $3(k_1+k_2)/k_1k_2\leq 6$, we obtain $\delta(M_{n\times n})=\mathcal{O}(1)$. On the other hand, for each $k\in[0\mathinner{.\,.}n-2]$ and $i\in[0\mathinner{.\,.}n-k-2]$, each factor $\mathtt{0}^i\mathtt{1}\mathtt{0}^k\mathtt{1}\mathtt{0}^{n-k-i-2}$ appears in $\mathtt{rlin}(M_{n\times n})$. There are $n-k-1$ of these factors for each k. Summing over all k, we obtain $P_M(n)/n=(n-1)/2=\Omega(n)$. Thus, $\delta(\mathtt{rlin}(M_{n\times n}))=\Omega(n)$, and the claim follows. □

Note that the same result holds trivially for b from the above proof, and a slightly weaker result can be shown for $g_{\mathtt{rl}}$ and g.

Proposition 8.6.2 There exists a family of square matrices $M_{n\times n}$ such that $b(M_{n\times n})=\mathcal{O}(1)$ and $b(\mathtt{rlin}(M_{n\times n}))=\Omega(n)$.

Proposition 8.6.3 There exists a family of square matrices $M_{n\times n}$ such that $\mu(M_{n\times n})=\mathcal{O}(\log n)$ and $\mu(\mathtt{rlin}(M_{n\times n}))=\Omega(n)$, where $\mu\in\{g,g_{\mathtt{rl}}\}$.

We do not conclude that linearizations are worthless, as their performance is highly dependent on the specific 2D string given. What we conclude is that we should keep looking for measures made specifically for 2D strings, which might or not include some kind of linearization step in between.

Chapter 9

Conclusion

Throughout this thesis, we have studied the phenomenon of repetitiveness in data, mostly from a theoretical point of view. To do so, we have studied combinatorial properties of well established state of the art repetitiveness measures, like the number of equal-letter runs in the Burrows-Wheeler transform of the text. We also designed and studied novel repetitiveness measures based on string morphisms and enhanced context-free grammars, in order to further understand the limits of current compression techniques used on highly repetitive texts. Moreover, we extended many repetitiveness measures from strings to work on two-dimensional data. Overall, we believe the results presented on this manuscript may shed some light on what needs to be added to state of the art compression techniques, in order to achieve better compression, or to work in more general datasets.

This chapter is structured as follows.

- In Section 9.1, we give an in-depth summary of the main contributions we have made on this thesis.
- In Section 9.2, we present some open questions and problems we left open for future work.

9.1 Summary of Contributions

We give a brief summary on the main contributions we have made through this thesis.

Chapter 5: Sensitivity Properties of the Burrows-Wheeler Transform

In the first part of Chapter 5, we introduced a string family w_k with $k \geq 5$, for which we can apply either an insertion, deletion or substitution, and increase the number of BWT-runs in the resulting word with respect to $r(w_k)$ by a $\Theta(\sqrt{n})$ additive factor. This substantially improves the known $\Omega(\log n)$ lower bound for additive sensitivity of r to all edit operations, recently proven by Akagi et al. [1]. Moreover, we showed the same result holds for the variant $r_\$$, and also proved that the difference between r and $r_\$$ can be $\Omega(\sqrt{n})$. These results in conjunction with the results on the multiplicative sensitivity of r [60], show that when working with the BWT, small changes to strings should be taken into account.

In the second part of Chapter 5, we studied the impact of morphism application on the number of BWT equal-letter runs of finite words. Firstly, we characterized Sturmian morphisms as the binary morphisms preserving the number of BWT equal-letter runs for all binary words containing both letters. Besides being interesting on its own, when this characterization is used in conjunction with the rest of our results, it allows us to construct binary words with any desired number of BWT-runs, and morphisms with known behavior. This can have practical applications, for instance, in experimentation. In fact, we showed an infinite family of binary morphisms called Thue–Morse-like morphisms, which increase the number of BWT-runs of binary words by 2. As a consequence, we have extended the results of Brlek et al. [20] on the number of BWT-runs of words generated by iterating the composition of the Fibonacci morphism with the Thue–Morse morphism to any composition of Sturmian morphisms and Thue–Morse-like morphisms. Also, we are able to construct infinite sequences of words of increasing length, having all exactly $2k$ BWT-runs, and converging to an aperiodic infinite word at their limit. While the result on Sturmian morphisms is a complete characterization, it is unknown if the compositions of Thue–Morse-like and Sturmian morphisms are the only binary morphisms increasing the number of BWT-runs exactly by 2. Then, we showed that when the alphabet size of the domain is $\sigma > 2$, the values $r(\varphi(w)) - r(w)$ and $r(\varphi(w))/r(w)$ can be arbitrarily large for some morphisms. In the case of the binary alphabet, we went further and showed that there exists morphisms where the additive sensitivity is $\Omega(\sqrt{n})$. Finally, we showed that the impact of morphism application on BWT-runs is quite different from the impact of morphisms on other repetitiveness measures based on popular compression schemes, like context-free grammars and LZ factorizations. In these measures, the additive increase after morphism application is bounded by a constant depending only on the morphism and the measure.

Chapter 6: New Repetitiveness Measures Based on Self-Similarity

We introduced a new repetitiveness measure we call ℓ, which exploits the self-similarity present on texts by representing them as the iteration of some string morphism over a starting symbol (modulo some other technicalities). This new measure can break the limits of δ —a measure considered a stable lower bound for repetitiveness— by a wide margin (a factor of $\Theta(\sqrt{n})$). On the other hand, ℓ can be asymptotically weaker than the space reached by several compressors based on run-length context-free grammars, many Lempel-Ziv variants, and the Burrows-Wheeler transform. Only the size of context-free grammars is an upper bound to ℓ. This suggests that the self-similarity exploited by L-systems is mostly independent of the source of repetitiveness exploited by other compressors and measures, which build on copy-paste mechanisms. We also show that several attempts to simplify or restrict L-systems lead to weaker measures.

In terms of improving compression, on the other hand, we introduced the measure ν, which aims to unify the repetitiveness induced by self-similarity and by explicit copies. This measure is the size of the smallest NU-system, a natural way to combine L-systems (with minimum size ℓ) with macro schemes (with minimum size $b \geq \delta$). In line with our finding that ℓ and δ are mostly orthogonal, we proved that ν is strictly more powerful than $\min(\ell, b)$, which makes ν one of the smallest reachable measures of repetitiveness to date.

In a more general perspective, Chapter 6 pushes a little further the discussion of what we understand by a repetitive string. Intuitively, repetitiveness is about copies, and macro schemes capture those copies pretty well, but there are other aspects in a text that could be repeated besides explicit copies, such as general patterns, or the relative ordering of symbols. Macro schemes capture explicit copies, L-systems capture so-called self-similarity, and NU-systems capture both.

Chapter 7: Extending Grammar-Based Measures

In Chapter 7, we have generalized a recent result by Ganardi et al. [52], which shows how to balance any SLP while maintaining its asymptotic size. Our generalization, called GSLP, allows in addition any rule of the form $A \rightarrow x$ where x is a program, of size $|x|$, that generates the actual (possibly much longer) right-hand side as long as every variable it contains appears at least twice. While we believe that this general result can be of wide interest to balance many kinds of generalizations of SLPs, we demonstrate its usefulness on a particular generalization of SLPs we call Iterated SLPs (ISLPs), which enable right-hand sides of the form $A \rightarrow \Pi_{i=k_1}^{k_2} B_1^{i^{c_1}} \cdots B_t^{i^{c_t}}$, of size $\mathcal{O}(t)$. We say a grammar is a d-ISLP when d is the maximum value of c_j along all those rules; we also call $g_{\texttt{it}}$ the size of the smallest ISLP that generates a given text. ISLPs are interesting in the context of compressibility measures for repetitive texts, as they are the first mechanism achieving size $\mathcal{O}(\delta/\sqrt{n})$ on some texts of length n —even with $d = 1$— while supporting polylogarithmic-time access —$\mathcal{O}(\log^2 n \log\log n)$— to arbitrary text symbols. This result is obtained thanks to the possibility of balancing the ISLP, and is extended to computing other substring queries like range minima and next/previous smaller value, which are useful for implementing suffix trees on repetitive text collections [50]. A further restriction, $d = \mathcal{O}(1)$, yields $\mathcal{O}(\log n)$ time for all the above queries, which is nearly optimal [128] for accessing the text in any grammar-compressed form. This class includes RLSLPs, which extend SLPs with the rule $A \rightarrow B^t$ and are equivalent to 0-ISLPs. We exploit again the possibility of balancing RLSLPs and show a technique to efficiently compute a wide class of substring queries we call "composable", that is, where $f(X \cdot Y)$ can be computed from $f(X)$ and $f(Y)$, e.g., Karp-Rabin fingerprints.

Chapter 8: Extending Repetitiveness Measures to the 2-Dimensional Space

In Chapter 8, we proposed the first complete extension of repetitiveness measures for one-dimensional strings to generic two-dimensional strings. In particular, we introduced extensions of the measures δ and γ to the two-dimensional case based on distinct factors of arbitrary rectangular shapes, as well as the extensions of the measures g, $g_{\texttt{rl}}$, and b, which are based on copy-paste mechanisms. We studied the mutual relationships between these measures and showed that $\delta \leq \gamma$ and $b \leq g_{\texttt{rl}} \leq g$. Further, we proved that, unlike in the 1D context where $\delta \leq \gamma \leq b \leq g_{\texttt{rl}} \leq g$, the two classes of measures become incomparable when

2D strings are considered. Indeed, depending on the 2D input, the measures g, $g_{\mathtt{rl}}$, and b can be asymptotically smaller than δ and γ. The results here presented suggest that in the 2D case, the measures δ and γ (as well as their square-based versions introduced in [22, 23]) are not satisfactory for capturing the regularities of a generic two-dimensional string, which are instead effectively detected by g, $g_{\mathtt{rl}}$, and b measures. At the same time, we have shown that the use of linearization strategies as preprocessing to compress two-dimensional input do not seem to be very effective.

9.2 Future Work

We present some open questions on the topics explored on this thesis. We also present some ideas for future research on other topics within the scope of repetitiveness measures, which we did not discuss in the previous chapters.

Chapter 5: Sensitivity Properties of the Burrows-Wheeler Transform

With respect to the sensitivity of BWT-runs to edit operations, there are still some directions for future investigation, mostly regarding if the bit catastrophes we have found could occur for string families with different asymptotic values of r, or if these are just anomalies that can happen in extremely specific cases. First, we ask whether there exist families of words with $r = \omega(1)$ for which edit operations can cause a multiplicative increase of $\Omega(\log n)$. Another interesting question is whether the upper bound $\mathcal{O}(r \log r \log n)$ from [1] for the additive sensitivity of r is tight. A weaker question is whether there exists an infinite family with $r = \omega(1)$ on which one edit operation can cause an additive increase of $\omega(r)$ in the number of runs of the BWT. All these questions are also valid for the variant $r_{\$}$.

Regarding the impact of morphisms on BWT-runs, we are working on some open questions we left. Probably the most interesting one is the following: what is a sufficient and necessary condition for a binary morphism φ, to have $r(\varphi(w)) - r(w) \leq 2k$ (for some $k > 0$ depending on φ) for every binary word w? Actually, we already have a characterization of these morphisms, to be published soon [43]. In this upcoming work we also show that for binary morphisms, $r(\varphi(w))/r(w)$ is always bounded by a constant depending only on φ.

In the future, we plan to study how to extend the results on morphism fixing BWT-runs, and morphisms increasing BWT-runs by a fixed natural number, to alphabets of size greater than 2, or to the variant $r_{\$}$.

Chapter 6: New Repetitiveness Measures Based on Self-Similarity

We left some open flanks on L-systems and NU-systems. Although we believe that the measures ℓ and ν are NP-hard to compute, this still needs to be proved. There is also the following question: is $\nu = \mathcal{O}(\gamma)$, or at least $o(\gamma \log(n/\gamma))$, for every string family? Recall that γ and $o(\gamma \log(n/\gamma))$ space is unknown to be reachable [71].

On the practical side of things, there is the question of whether we can approximate both measures, or answer common queries like `access`, `locate` and `count` efficiently in $\mathcal{O}(\ell)$ or $\mathcal{O}(\nu)$ space. For L-system, there are some works attempting to infer an L-system given an image or sequence [12, 62], which are based on genetic algorithms, though there are no guarantees on how these approaches would work on general sequences. In the case of NU-systems, even decompression seems non-trivial.

Finally, we believe that the variants ℓ_e and ℓ_p may be worthy of further study. They could be used for experiments and answering queries on prefixes of relevant families of infinite words, like morphic words and automatic words.

Chapter 7: Extending Grammar-Based Measures

Several questions remain open on ISLPs. One is about the cost to find the smallest ISLP that generates a given string w; we conjecture the problem is NP-hard as it is for plain SLPs and RLSLPs. Indeed, since RLSLPs correspond exactly to 0-ISLPs, finding the smallest 0-ISLP is NP-hard. It is open to extend this result to d-ISLPs for other values of d.

A second question is whether we can build an index within $\mathcal{O}(g_{\mathtt{it}})$ space that offers efficient pattern matching. While ISLPs support random access to the text, the typical path followed for SLPs [34] and for RLSLPs [32, App. A] cannot be directly applied for ISLPs, because iteration rules, which are of size $\mathcal{O}(t)$, would require indexing $\Theta(kt)$ positions. Computing Karp-Rabin fingerprints [66] on text substrings, which can be done in logarithmic time on SLPs and RLSLPs and enable substring equality and longest common prefix computations on w, is also challenging on general ISLPs.

Chapter 8: Extending Repetitiveness Measures to the 2-Dimensional Space

Regarding 2D repetitiveness measures, there are several open questions, as most of them were recently introduced. It has been shown that the 2D-Block Tree [19] —a data structure introduced for indexing 2D highly repetitive strings— can perform way worse than g on some relevant 2D string families [24]. This suggests it could be worthwhile to explore possible approximation strategies for b and g, as well as 2D versions of greedy grammar construction algorithms like the ones described in [8, 102].

We have make important advances regarding the query power of 2D SLPs. In fact, in the (currently under revision) journal version of our paper on 2D repetitiveness measures, we have shown how to provide access in $\mathcal{O}(g_{\mathtt{rl}})$ space and $\mathcal{O}(\log n)$ time. It is left open how to implement more complex queries like `locate` and `count`.

One can note that in the 1D setting, most repetitiveness measures can be upper bounded with respect to $\mathcal{O}(\delta \log(n/\delta))$, or another function of δ. Currently, there is not something like that for the 2D setting. Hence, we think it is important to find a meaningful combinatorial lower bound μ for b, g and $g_{\mathtt{rl}}$, so we can upper bound these 2D measures by a function of μ.

Finally, while we have shown some results on the issues of some commonly used linearization techniques for 2D strings, we believe it may be helpful to approach this topic in a more systematic and general manner.

Other Topics on Repetitiveness Measures

There are some interesting directions for future research on the general topic of repetitiveness measures.

Recently, Bannai et al. [5] showed that a certain restriction on Lempel-Ziv parsing that enables access to arbitrary positions of the text in $\mathcal{O}(\log n)$ predecessor queries, always reaches size $\mathcal{O}(g_{\texttt{rl}})$. The size $z^* \geq z$ of such parsing is then a new accessible repetitiveness measure that outperforms RLSLPs. The same string family that was used to show that z can be $o(g_{\texttt{rl}})$ serves to show that z^* can be $o(g_{\texttt{rl}})$; therefore z^* is a new accessible measure strictly better than $g_{\texttt{rl}}$ that should be further studied.

There is also the promising and recently introduced repetitiveness measure $\chi(w)$ defined as the size of the *smallest suffixient set* of a string [37]. It always holds that $\chi(w) \leq 2\min(r_{\$}(w), r_{\$}(w^R))$ [37], and it can be computed efficiently in linear time [29]. Because this measure has been recently introduced, there are many open question one can think of. For instance, what is the sensitivity of $\chi(w)$ to the most common string transformations? Or, can we generalize $\chi(w)$ to the 2-dimensional setting in a satisfactory manner? Moreover, the formulation of suffixient sets is interesting from a combinatorial point of view, similarly to the formulation of string attractors. Hence, we think studying how suffixient set can help to characterize relevant families of words (e.g., Sturmian words) can be an interesting line of work.

Bibliography

[1] T. Akagi, M. Funakoshi, and S. Inenaga. Sensitivity of string compressors and repetitiveness measures. *Information and Computation*, 291:104999, 2023.

[2] J.-P. Allouche, J. Cassaigne, J. Shallit, and L. Zamboni. A taxonomy of morphic sequences. *CoRR*, 1711.10807, 2017.

[3] J.-P. Allouche and J. Shallit. The ubiquitous Prouhet-Thue-Morse sequence. In *Sequences and their Applications*, pages 1–16. Springer London, 1999.

[4] J.-P. Allouche and J. Shallit. *Automatic Sequences - Theory, Applications, Generalizations*. Cambridge University Press, 2003.

[5] H. Bannai, M. Funakoshi, D. Hendrian, M. Matsuda, and S. Puglisi. Height-bounded Lempel-Ziv encodings. *CoRR*, 2403.08209, 2024.

[6] H. Bannai, M. Funakoshi, T. I, D. Köppl, T. Mieno, and T. Nishimoto. A separation of γ and b via Thue–Morse words. In *Proc. 28th International Symposium on String Processing and Information Retrieval (SPIRE 2021)*, volume 12944 of *Lecture Notes in Computer Science*, pages 167–178. Springer, 2021.

[7] H. Bannai, M. Funakoshi, K. Kurita, Y. Nakashima, K. Seto, and T. Uno. Optimal LZ-end parsing is hard. In *Proc. 34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023)*, volume 259 of *Leibniz International Proceedings in Informatics*, pages 3:1–3:11. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

[8] H. Bannai, M. Hirayama, D. Hucke, S. Inenaga, A. Jez, M. Lohrey, and C. Reh. The smallest grammar problem revisited. *IEEE Transactions on Information Theory*, 67(1):317–328, 2021.

[9] D. Belazzougui, F. Cunial, T. Gagie, N. Prezza, and M. Raffinot. Composite repetition-aware data structures. In *Proc. 26th Annual Symposium on Combinatorial Pattern Matching (CPM 2015)*, pages 26–39. Springer, 2015.

[10] T. Bell, J. Cleary, and I. Witten. *Text compression*. Prentice-Hall, Inc., USA, 1990.

[11] P. Berman, M. Karpinski, L. Larmore, W. Plandowski, and W. Rytter. On the complexity of pattern matching for highly compressed two-dimensional texts. *Journal of Computer and System Sciences*, 65(2):332–350, 2002.

[12] J. Bernard and I. McQuillan. Techniques for inferring context-free lindenmayer systems with genetic algorithm. *Swarm and Evolutionary Computation*, 64:100893, 2021.

[13] J. Berstel, D. Perrin, and C. Reutenauer. *Codes and Automata*, volume 129 of *Encyclopedia of mathematics and its applications*. Cambridge University Press, 2010.

[14] J. Berstel and P. Séébold. A characterization of Sturmian morphisms. In *Proc. 18th International Symposium on Mathematical Foundations of Computer Science (MFCS 1993)*, volume 711 of *Lecture Notes in Computer Science*, pages 281–290. Springer, 1993.

[15] V. Berthé, A. de Luca, and C. Reutenauer. On an involution of Christoffel words and Sturmian morphisms. *European Journal of Combinatorics*, 29(2):535–553, 2008.

[16] P. Bille, T. Gagie, I. Gørtz, and N. Prezza. A separation between RLSLPs and LZ77. *Journal of Discrete Algorithms*, 50:36–39, 2018.

[17] P. Bille, I. Gørtz, P. Cording, B. Sach, H. Vildhøj, and S. Vind. Fingerprints in compressed strings. *Journal of Computer and System Sciences*, 86:171–180, 2017.

[18] P. Bille, G. Landau, R. Raman, K. Sadakane, S. Satti, and O. Weimann. Random access to grammar-compressed strings and trees. *SIAM Journal on Computing*, 44(3):513–539, 2015.

[19] N. Brisaboa, T. Gagie, A. Gómez-Brandón, and G. Navarro. Two-dimensional block trees. *Computer Journal*, 67(1):391–406, 2024.

[20] S. Brlek, A. Frosini, I. Mancini, E. Pergola, and S. Rinaldi. Burrows-Wheeler transform of words defined by morphisms. In *Proc. 30th International Workshop on Combinatorial Algorithms (IWOCA 2019)*, volume 11638 of *Lecture Notes in Computer Science*, pages 393–404. Springer, 2019.

[21] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[22] L. Carfagna and G. Manzini. Compressibility measures for two-dimensional data. In *Proc. 30th International Symposium on String Processing and Information Retrieval (SPIRE 2023)*, volume 14240 of *Lecture Notes in Computer Science*, pages 102–113. Springer, 2023.

[23] L. Carfagna and G. Manzini. The landscape of compressibility measures for two-dimensional data. *IEEE Access*, 12:87268–87283, 2024.

[24] L. Carfagna, G. Manzini, G. Romana, M. Sciortino, and C. Urbina. Generalization of repetitiveness measures for two-dimensional strings. In *Proc. 31th International Symposium on String Processing and Information Retrieval (SPIRE 2024)*, volume 14899 of *Lecture Notes in Computer Science*, pages 57–72. Springer, 2024.

[25] J. Cassaigne. Sequences with grouped factors. In *Proc. 3rd International Conference on Developments in Language Theory (DLT 1997)*, pages 211–222. Aristotle University of Thessaloniki, 1997.

[26] J. Cassaigne, F. Gheeraert, A. Restivo, G. Romana, M. Sciortino, and M. Stipulanti. New string attractor-based complexities for infinite words. *Journal of Combinatorial Theory, Series A*, 208:105936, 2024.

[27] G. Castiglione, A. Restivo, and M. Sciortino. Hopcroft's Algorithm and Cyclic Automata. In *Proc. 2nd International Conference on Language and Automata Theory and Applications (LATA 2008)*, volume 5196 of *Lecture Notes in Computer Science*, pages 172–183. Springer, 2008.

[28] G. Castiglione, A. Restivo, and M. Sciortino. Circular Sturmian words and Hopcroft's algorithm. *Theoretical Computer Science*, 410(43):4372–4381, 2009.

[29] D. Cenzato, F. Olivares, and N. Prezza. On computing the smallest suffixient set. In *Proc. 31th International Symposium on String Processing and Information Retrieval (SPIRE 2024)*, volume 14899 of *Lecture Notes in Computer Science*, pages 73–87. Springer, 2024.

[30] G. Chaitin, A. Arslanov, and C. Calude. Program-size complexity computes the halting problem. *EATCS Bulletin*, 57, 1995.

[31] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

[32] A. R. Christiansen, M. B. Ettienne, T. Kociumaka, G. Navarro, and N. Prezza. Optimal-time dictionary-compressed indexes. *ACM Transactions on Algorithms*, 17(1):article 8, 2020.

[33] W.-F. Chuan. Sturmian morphisms and alpha-words. *Theoretical Computer Science*, 225(1-2):129–148, 1999.

[34] F. Claude, G. Navarro, and A. Pacheco. Grammar-compressed indexes with logarithmic search time. *Journal of Computer and System Sciences*, 118:53–74, 2021.

[35] S. Constantinescu and L. Ilie. The Lempel–Ziv complexity of fixed points of morphisms. *SIAM Journal on Discrete Mathematics*, 21(2):466–481, 2007.

[36] M. Crochemore, T. Lecroq, and W. Rytter. *125 Problems in Text Algorithms: with Solutions*. Cambridge University Press, 2021.

[37] L. Depuydt, T. Gagie, B. Langmead, G. Manzini, and N. Prezza. Suffixient sets. *CoRR*, abs/2312.01359, 2023.

[38] L. Dvořáková. String attractors of Episturmian sequences. *Theoretical Computer Science*, 986(C), 2024.

[39] A. Ehrenfeucht, K.P. Lee, and G. Rozenberg. Subword complexities of various classes of deterministic developmental languages without interactions. *Theoretical Computer Science*, 1(1):59–75, 1975.

[40] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. 41st IEEE Symposium on Foundations of Computer Science (FOCS 2000)*, pages 390–398. IEEE Computer Society, 2000.

[41] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):article 20, 2007.

[42] G. Fici, G. Romana, M. Sciortino, and C. Urbina. On the impact of morphisms on BWT-runs. In *Proc. 34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023)*, volume 259 of *Leibniz International Proceedings in Informatics*, pages 10:1–10:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[43] G. Fici, G. Romana, M. Sciortino, and C. Urbina. Morphisms and bwt-run sensitivity, 2025.

[44] J. Fischer and V. Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011.

[45] J. Fischer, V. Mäkinen, and G. Navarro. Faster entropy-bounded compressed suffix trees. *Theoretical Computer Science*, 410(51):5354–5364, 2009.

[46] M. Fredman and D. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.

[47] A. Frosini, I. Mancini, S. Rinaldi, G. Romana, and M. Sciortino. Logarithmic equal-letter runs for BWT of purely morphic words. In *Proc. 26th International Conference on Developments in Language Theory (DLT 2022)*, volume 13257 of *Lecture Notes in Computer Science*, pages 139–151. Springer, 2022.

[48] J. Fuchs and P. Whittington. The 2-Attractor Problem Is NP-Complete. In *Proc. 41st International Symposium on Theoretical Aspects of Computer Science (STACS 2024)*, volume 289 of *Leibniz International Proceedings in Informatics*, pages 35:1–35:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024.

[49] T. Gagie, G. Navarro, and N. Prezza. On the approximation ratio of Lempel-Ziv parsing. In *Proc. 13th Latin American Theoretical Informatics Symposium (LATIN 2018)*, volume 10807 of *Lecture Notes in Computer Science*, pages 490–503. Springer, 2018.

[50] T. Gagie, G. Navarro, and N. Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *Journal of the ACM*, 67(1):2:1–2:54, 2020.

[51] J. Gallant. *String Compression Algorithms.* PhD thesis, Princeton University, 1982.

[52] M. Ganardi, A. Jez, and M. Lohrey. Balancing straight-line programs. *Journal of the ACM*, 68(4):27:1–27:40, 2021.

[53] L. Gasieniec, M. Karpinski, W. Plandowski, and W. Rytter. Efficient algorithms for Lempel-Ziv encoding. In *Proc. 5th Scandinavian Workshop on Algorithm Theory (SWAT 1996)*, volume 1097 of *Lecture Notes in Computer Science*, pages 392–403, 1996.

[54] L. Gasieniec, R. Kolpakov, I. Potapov, and P. Sant. Real-time traversal in grammar-based compressed files. In *Proc. 15th Data Compression Conference (DCC 2005)*, pages 458–, 2005.

[55] F. Gheeraert, G. Romana, and M. Stipulanti. String attractors of fixed points of k-bonacci-like morphisms. In *Proc. 14th International Conference on Combinatorics on Words (WORDS 2023)*, volume 13899 of *Lecture Notes in Computer Science*, pages 192–205. Springer, 2023.

[56] F. Gheeraert, G. Romana, and M. Stipulanti. String attractors of some simple-parry automatic sequences. *Theory of Computing Systems*, 2024.

[57] D. Giammarresi and A. Restivo. Two-dimensional languages. In *Handbook of Formal Languages (3)*, pages 215–267. Springer, 1997.

[58] R. Giancarlo. A generalization of the suffix tree to square matrices, with applications. *SIAM Journal on Computing*, 24(3):520–562, 1995.

[59] S. Giuliani, S. Inenaga, Z. Lipták, N. Prezza, M. Sciortino, and A. Toffanello. Novel results on the number of runs of the Burrows-Wheeler-transform. In *Proc. 47th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2021)*, pages 249–262, 2021.

[60] S. Giuliani, S. Inenaga, Z. Lipták, G. Romana, M. Sciortino, and C. Urbina. Bit catastrophes for the Burrows-Wheeler transform. In *Proc. 27th International Conference on Developments in Language Theory (DLT 2023)*, volume 13911 of *Lecture Notes in Computer Science*, pages 86–99. Springer, 2023.

[61] S. Giuliani, S. Inenaga, Z. Lipták, G. Romana, M. Sciortino, and C. Urbina. Bit catastrophes for the burrows-wheeler transform. *Theory of Computing Systems*, 69(2):19, 2025.

[62] J. Guo, H. Jiang, B. Benes, O. Deussen, X. Zhang, D. Lischinski, and H. Huang. Inverse procedural modeling of branching structures by inferring l-systems. *ACM Transactions on Graphics*, 39(5), 2020.

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

[64] E. Kaltofen and G. Villard. On the complexity of computing determinants. *Computational Complexity*, 13:91–130, 2004.

[65] J. Karhumäki. On cube-free ω-words generated by binary morphisms. *Discrete Applied Mathematics*, 5(3):279–297, 1983.

[66] R. Karp and M. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[67] A. Kawamoto, T. I, D. Köppl, and H. Bannai. On the hardness of smallest RLSLPs and collage systems. In *Proc. 34th Data Compression Conference (DCC 2024)*, pages 243–252, 2024.

[68] D. Kempa and T. Kociumaka. Resolution of the Burrows-Wheeler transform conjecture. *Communications of the ACM*, 65(6):91–98, 2022.

[69] D. Kempa and T. Kociumaka. Collapsing the hierarchy of compressed data structures: Suffix arrays in optimal compressed space. In *Proc. 64th IEEE Annual Symposium on Foundations of Computer Science (FOCS 2023)*, pages 1877–1886, 2023.

[70] D. Kempa and D. Kosolobov. LZ-end parsing in linear time. In *Proc. 25th Annual European Symposium on Algorithms (ESA 2017)*, volume 87 of *Leibniz International Proceedings in Informatics*, pages 53:1–53:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017.

[71] D. Kempa and N. Prezza. At the roots of dictionary compression: String attractors. In *Proc. 50th Annual ACM SIGACT Symposium on Theory of Computing (STOC 2018)*, page 827–840. Association for Computing Machinery, 2018.

[72] D. Kempa and B. Saha. An upper bound and linear-space queries on the LZ-end parsing. In *Proc. 33rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2022)*, pages 2847–2866, 2022.

[73] T. Kida, T. Matsumoto, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. Collage system: A unifying framework for compressed pattern matching. *Theoretical Computer Science*, 298(1):253–272, 2003.

[74] J. Kieffer and E.-H. Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, 2000.

[75] D. Kim, Y. Kim, and K. Park. Generalizations of suffix arrays to multi-dimensional matrices. *Theoretical Computer Science*, 2003.

[76] D. Knuth. Johann faulhaber and sums of powers. *Mathematics of Computation*, 61(203):277–294, 1993.

[77] D. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[78] T. Kociumaka, G. Navarro, and F. Olivares. Near-optimal search time in δ-optimal space, and vice versa. *Algorithmica*, 86(4):1031–1056, 2024.

[79] T. Kociumaka, G. Navarro, and N. Prezza. Towards a definitive compressibility measure for repetitive sequences. *IEEE Transactions on Information Theory*, 69(4):2074–2092, 2023.

[80] S. Kreft and G. Navarro. LZ77-like compression with fast random access. In *Proc. 20th Data Compression Conference (DCC 2010)*, pages 239–248, 2010.

[81] S. Kreft and G. Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.

[82] J. Shallit L. Schaeffer. String attractors for automatic sequences. *CoRR*, abs/2012.06840, 2020.

[83] B. Langmead and S. Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature Methods*, 9(4):357–359, 2012.

[84] N. Larsson and A. Moffat. Off-line dictionary-based compression. *Procedings of the IEEE*, 88(11):1722–1732, 2000.

[85] A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.

[86] A. Lempel and J. Ziv. Compression of two-dimensional data. *IEEE Transactions on Information Theory*, 32(1):2–8, 1986.

[87] H. Li and R. Durbin. Fast and accurate long-read alignment with Burrows–Wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.

[88] A. Lindenmayer. Mathematical models for cellular interactions in development I. Filaments with one-sided inputs. *Journal of Theoretical Biology*, 18(3):280–299, 1968.

[89] A. Lindenmayer. Mathematical models for cellular interactions in development II. Simple and branching filaments with two-sided inputs. *Journal of Theoretical Biology*, 18(3):300–315, 1968.

[90] M. Lothaire. *Algebraic Combinatorics on Words*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, New York, NY, USA, 2002.

[91] R. Lyndon and M-P. Schützenberger. The equation $a^m = b^n c^p$ in a free group. *Michigan Mathematical Journal*, 9(4):289–298, 1962.

[92] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[93] S. Mantaci, A. Restivo, G. Romana, G. Rosone, and M. Sciortino. A combinatorial view on string attractors. *Theoretical Computer Science*, 850:236–248, 2021.

[94] S. Mantaci, A. Restivo, G. Rosone, M. Sciortino, and L. Versari. Measuring the clustering effect of BWT via RLE. *Theoretical Computer Science*, 698:79–87, 2017.

[95] S. Mantaci, A. Restivo, and M. Sciortino. Burrows–Wheeler transform and Sturmian words. *Information Processing Letters*, 86(5):241–246, 2003.

[96] G. Manzini. An analysis of the Burrows–Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.

[97] F. Mignosi and P. Séébold. Morphismes Sturmiens et règles de Rauzy. *Journal de théorie des nombres de Bordeaux*, 5(2):221–233, 1993.

[98] J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Succinct representations of permutations and functions. *Theoretical Computer Science*, 438:74–88, 2012.

[99] G. Navarro. Indexing highly repetitive string collections, part I: Repetitiveness measures. *ACM Computing Surveys*, 54(2):article 29, 2021.

[100] G. Navarro. Indexing highly repetitive string collections, part II: Compressed indexes. *ACM Computing Surveys*, 54(2):article 26, 2021.

[101] G. Navarro. The compression power of the BWT: technical perspective. *Communications of the ACM*, 65(6):90, 2022.

[102] G. Navarro, C. Ochoa, and N. Prezza. On the approximation ratio of ordered parsings. *IEEE Transactions on Information Theory*, 67(2):1008–1026, 2021.

[103] G. Navarro, F. Olivares, and C. Urbina. Balancing run-length straight-line programs. In *Proc. 29th International Symposium on String Processing and Information Retrieval (SPIRE 2022)*, volume 13617 of *Lecture Notes in Computer Science*, pages 117–131. Springer, 2022.

[104] G. Navarro, F. Olivares, and C. Urbina. Generalized straight-line programs. *Acta Informatica*, 62(1):14, 2025.

[105] G. Navarro and C. Urbina. On stricter reachable repetitiveness measures. In *Proc. 28th International Symposium on String Processing and Information Retrieval (SPIRE 2021)*, volume 12944 of *Lecture Notes in Computer Science*, pages 193–206. Springer, 2021.

[106] G. Navarro and C. Urbina. L-systems for measuring repetitiveness. In *Proc. 34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023)*, volume 259 of *Leibniz International Proceedings in Informatics*, pages 25:1–25:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[107] G. Navarro and C. Urbina. Iterated straight-line programs. In *Proc. 16th Latin American Theoretical Informatics Symposium (LATIN 2024)*, volume 14578 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2024.

[108] G. Navarro and C. Urbina. Repetitiveness measures based on string morphisms. *Theoretical Computer Science*, 1043:115259, 2025.

[109] Y. Nekrich and G. Navarro. Sorted range reporting. In *Proc. 13th Scandinavian Symposium on Algorithmic Theory (SWAT 2012)*, volume 7357 of *Lecture Notes in Computer Science*, pages 271–282, 2012.

[110] T. Nishimoto, T. I, S. Inenaga, H. Bannai, and M. 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*, pages 72:1–72:15, 2016.

[111] J.-J. Pansiot. On various classes of infinite words obtained by iterated mappings. In *Proc. 1st LITP Spring School on Theoretical Computer Science (LITP 1984)*, volume 192 of *Lecture Notes in Computer Science*, pages 188–197, 1984.

[112] J.-J. Pansiot. Subword complexities and iteration. *Bulletin of the European Association for Theoretical Computer Science*, 26:55–62, 1985.

[113] G. Paquin. On a generalization of Christoffel words: Epichristoffel words. *Theoretical Computer Science*, 410(38-40):3782–3791, 2009.

[114] H. Petersen. On the language of primitive words. *Theoretical Computer Science*, 161(1):141–156, 1996.

[115] M. Przeworski, R. Hudson, and A. Di Rienzo. Adjusting the focus on human variation. *Trends in Genetics*, 16(7):296—302, 2000.

[116] S. Raskhodnikova, D. Ron, R. Rubinfeld, and A. Smith. Sublinear algorithms for approximating string compressibility. *Algorithmica*, 65(3):685–709, 2013.

[117] A. Restivo and G. Rosone. Balanced words having simple Burrows–Wheeler transform. In *Proc. 13th International Conference on Developments in Language Theory (DLT 2009)*, volume 5583 of *Lecture Notes in Computer Science*, pages 431–442. Springer, 2009.

[118] G. Romana. Algorithmic view on circular string attractors. In *Proc. 24th Italian Conference on Theoretical Computer Science (ICTCS 2023)*, volume 3587 of *CEUR Workshop Proceedings*, pages 169–180. CEUR-WS.org, 2023.

[119] G. Romana. *Repetitiveness Measures based on String Attractors and Burrows-Wheeler Transform: Properties and Applications*. PhD thesis, Università degli Studi Palermo, 2023.

[120] G. Rozenberg and A. Salomaa. *The mathematical theory of L systems*. Academic press, 1980.

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

[122] H. Sagan. *Space-Filling Curves*. Universitext. Springer, New York, NY, 1994.

[123] A. Salomaa. On exponential growth in Lindenmayer systems. *Indagationes Mathematicae (Proceedings)*, 76(1):23–30, 1973.

[124] M. Sciortino and L. Zamboni. Suffix automata and standard Sturmian words. In *Proc. 11th International Conference on Developments in Language Theory (DLT 2007)*, volume 4588 of *Lecture Notes in Computer Science*, pages 382–398. Springer, 2007.

[125] J. Shallit and D. Swart. An efficient algorithm for computing the ith letter of $\varphi^n(a)$. In *Proc. 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1999)*, pages 768–775, 1999.

[126] M. Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012.

[127] J. Storer and T. Szymanski. Data compression via textual substitution. *Journal of the ACM*, 29(4):928–951, 1982.

[128] E. Verbin and W. Yu. Data structure lower bounds on random access to grammar-compressed strings. In *Proc. 24th Annual Symposium on Combinatorial Pattern Matching (CPM 2013)*, volume 7922 of *Lecture Notes in Computer Science*, pages 247–258, 2013.