Repetitiveness Measures Based on String Morphisms

Gonzalo Navarro, Cristian Urbina*

Department of Computer Science, University of Chile, Chile

Centre for Biotechnology and Bioengineering (CeBiB), Chile

Abstract

We study deterministic Lindenmayer systems, and more generally string morphisms, as mechanisms to capture the repetitiveness of string collections. In particular, we define L-systems, which extend CPD0L-systems with two parameters, d and n, to unambiguously determine a single string $w = \tau(\varphi^d(s))[1\,..\,n]$, where φ is the morphism of the system, τ is a coding, and s is its starting symbol. We define $\ell(w)$ as the size of the shortest description of an L-system generating w, and show that ℓ, which builds on the self-similarities that arise in the sequence, is a relevant measure of repetitiveness.

We study the relation between ℓ and a better-established measure called δ, which builds on substring complexity. Our results show that ℓ and δ are largely orthogonal, in the sense that either can be larger than the other, depending on the string family. In particular, ℓ can be $\Theta(\sqrt{n})$ times smaller than δ, whereas ℓ is upper-bounded by the smallest context-free grammar that generates the string, and thus can be only polylogarithmically larger than δ. This suggests that both mechanisms capture different kinds of regularities related to repetitiveness.

We then combine the mechanisms of L-systems and of bidirectional macro schemes, the smallest known representation scheme that captures substring copies. We call the combination NU-systems, and show that they can be asymptotically strictly smaller than both mechanisms for the same fixed string family. The size ν of the smallest NU-system is a new minimal reachable repetitiveness measure, and it is also shown to be uncomparable with δ. This shows that combining morphism substitution with copy-paste mechanisms yields a stronger approach to capture string repetitiveness.

We also study algorithmic problems on L-systems and NU-systems, such as the cost of decompression and of accessing substrings of the encoded string without fully decompressing it, and explore various simplifications of L-systems that trade compression effectiveness by algorithmic efficiency to handle them.

Keywords: Repetitiveness measures, Text compression, Compact data

*Corresponding author

Email addresses: gnavarro@dcc.uchile.cl (Gonzalo Navarro), crurbina@dcc.uchile.cl (Cristian Urbina)

1. Introduction

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 [32]. 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.

A strong theoretical measure of string repetitiveness introduced by Kociumaka et al. [18] is δ, based on the substring complexity function. This measure has several nice properties: it is computable in linear time, monotone, resistant to string edits, insensitive to simple string transformations, and it lower-bounds almost every other theoretical or *ad-hoc* repetitiveness measure considered in the literature. Further, although $\mathcal{O}(\delta)$ space is unreachable (i.e., one cannot encode every string in $\mathcal{O}(\delta)$ space), there exist $\mathcal{O}(\delta \log(n/\delta))$-space representations that in addition support efficient pattern matching queries [17, 18], and this space is tight: no $o(\delta \log(n/\delta))$-space representation can exist [18].

The idea that δ is a sound lower bound for repetitiveness is reinforced by the fact that it is always $\mathcal{O}(b)$, where b is the size of the smallest *bidirectional macro scheme (BMS)* generating a string [37]. Those macro schemes arguably capture every possible way of exploiting copy-paste regularities in the sequences.

In this paper we explore a source of repetitiveness that is more structural and captured by *string morphisms*, and show that it can sharply break the lower bound given by δ. The simplest system we propose, which we call *L-system*, builds upon deterministic Lindenmayer systems [21, 22], 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.

Since ℓ is a reachable measure of repetitiveness (because the L-system is a representation of w of size $\mathcal{O}(\ell)$), there are string families where $\delta = o(\ell)$.

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

Intriguingly, we show that ℓ is as small as $\mathcal{O}(\delta/\sqrt{n})$ on other string families, so (1) both measures are uncomparable and (2) the lower bound δ does not capture this kind of repetitiveness. On the other hand, we show that $\ell = \mathcal{O}(g)$, where g is the size of the smallest deterministic context-free grammar generating only w. 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. Indeed, L-systems are similar to grammars, differing in that they have no terminal symbols, so their expansion must be explicitly stopped at level d and then possibly converted to terminals with τ.

We also introduce NU-systems, which combine the power of L-systems with BMSs. The measure $\nu(w)$, defined as the size of the smallest NU-system generating w, becomes a new minimal reachable repetitiveness measure. We show that the smallest NU-system generating some string families strictly lower bounds both the smallest L-system and the smallest BMS.

After presenting the basic concepts in Section 2, our concrete contributions in the paper are the following:

1. In Section 3, we introduce L-systems as compression devices. We also introduce the measure ℓ, defined as the size of the smallest L-system generating the string. We show how to perform basic operations like decompression or direct access on L-systems, and prove that they are monotone to the appending of prefixes.
2. In Section 4, 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. On the other hand, we show that ℓ is always in $\mathcal{O}(g)$, which places ℓ in the map of known repetitiveness measures.
3. In Section 5, we expose string families where ℓ is larger than the output of several repetitiveness-aware compressors like the size g_{rl} of the smallest run-length context-free grammar, the size z_e of the greedy LZ-End parse [19], and the number of runs r in the Burrows-Wheeler Transform of the string [4]. We then conclude that ℓ is uncomparable to almost all measures other than g, which suggests that the source of repetitiveness it captures is largely orthogonal to the typical cut-and-paste of macro schemes.
4. In Section 6, we introduce *macro-systems*, which are a reformulation of BMSs, 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.
5. In Section 7, 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.

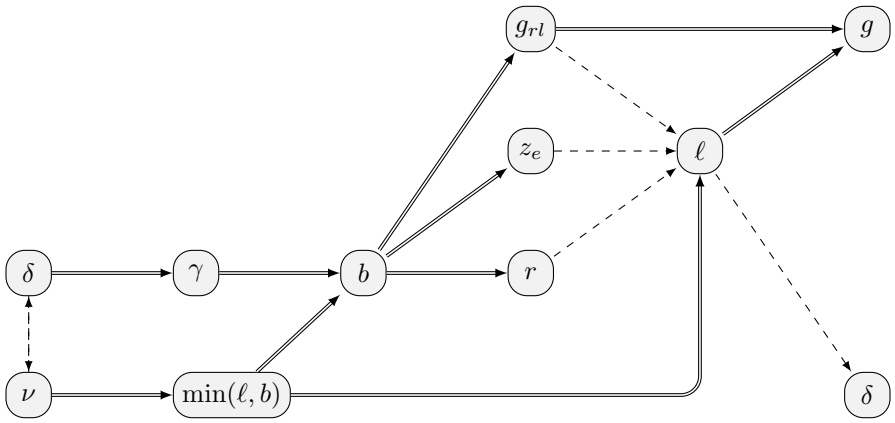

Figure 1: Asymptotic relations between ℓ, ν, and other repetitiveness measures. A double solid arrow from v_1 to v_2 means that it always holds that $v_1 = \mathcal{O}(v_2)$, and there exists a string family where $v_1 = o(v_2)$. A dashed arrow from v_1 to v_2 means that there exists a family where $v_1 = o(v_2)$.

6. In Section 8, 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.

Finally, in Section 9 we present the conclusions and some open question regarding L-systems, NU-systems, and alternative ways to exploit repetitiveness.

Overall, our results contribute to understanding how to measure repetitiveness and how to exploit it in order to build better compressors. Figure 1 shows how our new measures ℓ and ν relate to others in the literature.

This work is an extended version of articles published in SPIRE 2021 [26] and CPM 2023 [27]. In this article, we improve various results and obtain several new ones, give more detailed proofs, and provide a unified perspective.

2. Terminology

In this section we explain the basic terminology and concepts needed to understand the rest of the paper, from strings and morphisms to relevant repetitiveness measures.

2.1. Strings

An *alphabet* is a finite set of *symbols* $\Sigma = \{a_1, \ldots, a_{|\Sigma|}\}$. An *ordered alphabet* is an alphabet extended with a total order $a_1 < \cdots < a_{|\Sigma|}$. A (finite) *string* w is a finite 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^*. 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 $x[1]\cdots x[n]\,y[1]\cdots y[m]$. 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 $w[i\,..\,j] = w[1\,..\,j]$ if $i < 1$, $w[i\,..\,j] = w[i\,..\,n]$ if $j > n$, and $w[i\,..\,j] = \varepsilon$ if $i > n$ or $j < 1$. The *reverse* of w, denoted w^R, is the string $w[|w|]\cdot w[|w|-1]\cdots w[1]$.

A (right) *infinite string* $\mathbf{w}$ (we use boldface to emphasize them) over an alphabet Σ is a mapping from $\mathbb{Z}^+$ to Σ, and its length is called ω, which is greater than n for any $n \in \mathbb{Z}^+$. The concatenation $x\cdot\mathbf{y}$ is 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.

2.2. 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 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*.

Let $\varphi : \Sigma_1^* \to \Sigma_2^*$ be a morphism on strings. Some useful definitions are $\mathit{width}(\varphi) = \max_{a\in\Sigma_1} |\varphi(a)|$ 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 $\forall a \in \Sigma_1, |\varphi(a)| = 1$ (sometimes called a 1-uniform morphism).

Let $\varphi : \Sigma^* \to \Sigma^*$ be an endomorphism. Then φ is *prolongable* on a symbol a if $\varphi(a) = ax$ for some string $x \neq \varepsilon$. If this is the case, then for each i, j with $0 \leq i \leq j$, it holds that $\varphi^i(a)$ is a prefix of $\varphi^j(a)$, and $\mathbf{x} = \varphi^\omega(a) = ax\varphi(x)\varphi^2(x)\cdots$ is the unique infinite fixed-point of φ starting with the symbol a. An infinite string $\mathbf{w} = \varphi^\omega(a)$ that is the fixed-point of a morphism is called a *purely morphic word*, its image under a coding $\mathbf{x} = \tau(\mathbf{w})$ is called a *morphic word*, and if the morphism φ is k-uniform, then $\mathbf{x}$ is said to be *k-automatic*.

2.3. Repetitiveness measures

A *repetitiveness measure* μ is a function from strings to numbers that 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.

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). Space is usually measured 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 $d \geq 0$.

A repetitiveness measure μ_1 *is smaller* or *lower-bounds* another 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 repetitiveness measures μ_1 and μ_2 are *equivalent* if each one is smaller than the other, and *uncomparable* if none is (i.e., $\mu_1 = o(\mu_2)$ on a string family $\mathcal{F}_1$ and $\mu_2 = o(\mu_1)$ on another string family $\mathcal{F}_2$).

In the following, we explain the most relevant repetitiveness measures to be considered in the rest of the paper. All of them, 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, $\mathcal{O}(n)$. For a more in-depth review, see a recent survey [24].

2.3.1. 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 symbol*. For readability, we write the rules (A, x) as $A \to x$. Let $u, v \in (V \cup \Sigma)^*$. If $A \to x$ 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 $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$, that is, the strings of terminals that can be derived from the initial symbol.

A *straight-line program (SLP)* is a CFG G such that any rule is either a *terminal rule* $A \to a$ with $a \in \Sigma$, or a *binary rule* $A \to BC$ 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 $A \to BC$, then $B, C < A$. In this paper we will only consider CFGs that admit such a total order, though the right-hand sides of rules may have zero or more terminals and variables. That is, if $A \to \alpha B \beta$ is a rule, then $B < A$. Such CFGs are guaranteed to generate a unique string, which is denoted $\exp(G) = w$. 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 $\texttt{size}(G) = \sum\{|x| \mid A \to x \in R\}$, 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 [34, 5], though there exist log-approximations [12, 34].

Another measure related to CFGs that strictly lower-bounds $g(w)$ is $g_{rl}(w)$, the least size of a *run-length CFG (RLCFG)* generating w [29]. RLCFGs extend

CFGs by allowing constant-size rules of the form $A \to B^k$ for $k > 1$ and $B \in V \cup \Sigma$, 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 $g_{rl} = \mathcal{O}(1)$.

Composition-systems [9] extend CFGs with constant-size *extraction* rules of the form $A \to B[i:j]$ for some $i, j \in [1, |\exp(B)|]$, which mean that $\exp(A) = \exp(B)[i\,..\,j]$. Still, the symbols must be ordered and it must hold $B < A$ for such a rule to be valid. *Collage-systems* [16] 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 strictly lower-bounds $g_{rl}(w)$.

2.3.2. Parsing-based measures

A *parsing* of size k produces 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 [20] 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 [19] requires, in addition, that the source ends at a previous phrase boundary. All of these parsings can be constructed in linear time, and their number of phrases are denoted by z, z_{no}, and z_e, respectively. While z and z_{no} are optimal among the parsings satisfying their respective conditions, this is not always the case for z_e. The optimal factorization where each phrase w_{i+1} appears as a suffix of $w_1 \dots w_j$ for some $j \leq i$ is denoted by z_{end}. Because of the optimality of z, z_{no}, and z_{end}, it holds that $z \leq z_{no} \leq z_{end} \leq z_e$ for every string.

A *bidirectional macro scheme* (BMS) [37] 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 [25], except for the ones we define in this paper. Computing $b(w)$ is NP-hard [7].

Another interesting parsing-based measure is the size of the greedy *lexicographic parsing* of w, denoted as $v(w)$ [25]. 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 a lexicographically smaller suffix starting inside the processed part (a unique symbol $\mathtt{\$}$, smaller than the others, is assumed to exist at the end of w). It forms an explicit phrase of length 1 if the longest common prefix is empty or no predecessor exists.

2.3.3. Burrows-Wheeler transform

The Burrows-Wheeler transform (BWT) [4] is a reversible transformation that usually makes a string more compressible. It is obtained by concatenating the symbols preceding the sorted suffixes of $w\$$. The BWT tends to produce long runs of the same symbol a when a string is repetitive, and these (maximal) runs of length k can be encoded as pairs (a, k). A repetitiveness measure, $r(w)$, based on this encoding is then the number of maximal runs in the BWT of w. Although r is not ideal as a repetitiveness measure (e.g., its size can vary significantly upon minor changes in the string) [11], its size can be bounded in terms of z [14].

2.3.4. String attractors: the measure γ

Kempa and Prezza [15] introduced the notion of *string attractor* as a unifying framework and lower bound for grammar-based and parsing-based compressors.

A string attractor for a string $w[1\,..\,n]$ 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, every substring of w has a copy covering a position in Γ. The measure $\gamma(w)$ is defined as the size of the smallest string attractor for w.

Computing $\gamma(w)$ is NP-hard. The measure γ strictly lower-bounds b [2]. Yet, it is unknown whether space γ, or even $o(\gamma \log(n/\gamma))$, is reachable.

2.3.5. Substring complexity: the measure δ

Let $F_w(k)$ be the set of distinct substrings of $w[1\,..\,n]$ of length k, $F_w(k) = \{w[i\,..\,i+k-1] \mid 1 \leq i \leq n-k+1\}$. The *substring complexity function* of w is defined as $P_w(k) = |F_w(k)|$. Kociumaka et al. [18] introduced a repetitiveness measure based on this function, defined as $\delta(w) = \max\{P_w(k)/k \mid k \in [1\,..\,n]\}$.

The measure δ strictly lower-bounds almost every other repetitiveness measure considered in the literature, including γ. It also has several nice properties: it is computable in linear time, monotone, insensitive to reversals, resistant to small edits on w, and can be used to construct $\mathcal{O}(\delta \log(n/\delta))$-space representations supporting efficient access and pattern matching queries [18, 17]. On the other hand, $o(\delta \log(n/\delta))$ space is known to be unreachable [18].

A recent mechanism called ISLPs [28] was proposed with the aim of showing that there are reachable measures, which can even be efficiently accessed at arbitrary positions, that can break the lower bound given by δ. We believe that the measures we propose here are more natural and of wider interest.

3. Deterministic L-systems and the Measure ℓ

In this section we study a mechanism for generating infinite sequences called *deterministic Lindenmayer systems* (L-systems) [21, 22], which build on string morphisms. L-systems were initially utilized as a tool to model the growth of plants and algae [21, 22]. They also have been used to define infinite words with interesting self-similarity and factor complexity properties [33]. 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 $w[1\,..\,n]$ 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 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 $w = \tau(\varphi^d(s))[1\,..\,n]$.*

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

Definition 2 (Measure ℓ)**.** *The* size *of an L-system $L = (\Sigma, \varphi, \tau, s, d, n)$ is $\mathtt{size}(L) = \mathtt{size}(\varphi) + |\Sigma| + 3$. 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 $\mathcal{O}(\mathtt{size}(L))$ space. Hence, the measure ℓ is reachable. As a convention, we always assume that d and $|\Sigma|$ are in $n^{\mathcal{O}(1)}$. Otherwise, our RAM machine would need $\omega(1)$ $\Theta(\log n)$-bit words to represent the integer d or the symbols of the alphabet, and could not operate them in constant time. We also assume that $\mathtt{size}(L) \leq 3n$ (and hence $|\Sigma| \leq 3n$), as there is always a trivial L-system $L = (\Sigma', \{s \to w\}, \mathtt{id}, s, 1, n)$ generating w, of size $n + |\Sigma'| + 3$, where Σ' contains the symbols actually appearing in w, and $\mathtt{id}$ 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.

3.1. Decompression

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

Algorithm 1: 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 $n - n'$.

```
1 function decompress(a, d, n)
2     if d = 0 then
3         output τ(a)
4         return n − 1
5     let a → b_1 ··· b_k ∈ φ
6     for i ← 1 to k do
7         n ← decompress(b_i, d − 1, n)
8         if n = 0 then return 0
9     return n
```

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 1, 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 $h \geq 0$ [23]. We also define function g as $g(a) = \min\big(\{h \geq 0 \mid f^h(a) \text{ is not unary}\} \cup \{+\infty\}\big)$. 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, $f(a)$, $f^2(a)$, ... 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 2 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 $|\Sigma| = \mathcal{O}(n)$.

Theorem 1. *Given an L-system $L = (\Sigma, \varphi, \tau, s, d, n)$, we can compute its represented string in time $\mathcal{O}(n + d)$.*

Algorithm 2: 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 $n - n'$.

```
1  function decompress(a, d, n)
2      h ← g(a)
3      if h ≥ d then
4          output τ(f^d(a))
5          return n − 1
6      b ← f^h(a)
7      let b → b_1 ··· b_k ∈ φ
8      for i ← 1 to k do
9          n ← decompress(b_i, d − h − 1, n)
10         if n = 0 then return 0
11     return n
```

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 [10] 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 $b \to a\, b_2 \cdots b_k$. We recursively decompress $b_2, \ldots, b_k$ with $d = 0$. Now let $c = f^{d-2}(a)$ and $c \to b\, c_2 \cdots c_r$. 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 3. 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 $L = (\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, \{\mathtt{a} \to \mathtt{ba}, \mathtt{b} \to \mathtt{c}, \mathtt{c} \to \mathtt{b}\}, \tau, \mathtt{a}, d, n)$ with large d.

Theorem 2. *Given an L-system $L = (\Sigma, \varphi, \tau, s, d, n)$, we can compute its represented string in time $\mathcal{O}(n \log d)$.*

3.2. 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 [3], but not on Lempel-Ziv-like parsings.

Algorithm 3: Decompressing L-systems $L = (\Sigma, \varphi, \tau, s, d, n)$ in time $\mathcal{O}(n \log d)$; 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 $n - n'$.

```
1  function decompress(a, d, n)
2      output τ(f^d(a))
3      n ← n − 1
4      if n = 0 then return 0
5      h ← d − 1
6      while h ≥ 0 do
7          b ← f^h(a)
8          let b → b_1 ··· b_k ∈ φ
9          for j ← 2 to k do
10             n ← decompress(b_j, d − h − 1, n)
11             if n = 0 then return 0
12         h ← max({k ∈ [0..h−1] | k + g(f^k(a)) < h} ∪ {−1})  (binsearch)
13     return n
```

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 $a \to b_1 \cdots b_k \in \varphi$, then we define $p_a^l(t) = \sum_{r=1}^{t} |b_r^l|$ for $0 \le t \le k$.

We begin the extraction of $w[i]$ from the axiom $a_0 = s$, with $i_0 = i$. Let $r_0 \ge 1$ be such that $p_{a_0}^d(r_0 - 1) < i_0 \le p_{a_0}^d(r_0)$, and $a_0 \to b_1 \cdots b_k \in \varphi$. 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 4 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 [3] 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 $i_r < x \le i_{r+1}$. 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 $n \le |\varphi|^d$, 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 $|\Sigma| \times |\Sigma|$ 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. [35]). Formally, if $a \to b_1 \cdots b_k \in \varphi$, 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 $\mathcal{O}(d(|\varphi| + |\Sigma|)) = \mathcal{O}(d|\varphi|)$ total time and $\mathcal{O}(|\varphi| + d|\Sigma|)$ space. Once the vectors L_l are obtained, we can compute the functions p_a^l in $\mathcal{O}(d|\varphi|)$ space and time.

Algorithm 4: 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 ∈ Σ, l ∈ [0 .. d − 1]
3     a ← s
4     for l ← d downto 1 do
5         let a → b_1 ··· b_k ∈ φ
6         let r be such that p_a^l(r − 1) < i ≤ p_a^l(r) (interval-biased search)
7         i ← i − p_a^l(r − 1)
8         a ← b_r
9     return τ(a)
```

The interval-biased search trees are built in linear time and space, which adds up to $\mathcal{O}(d|\varphi|)$ in our case.

Theorem 3. *After $\mathcal{O}(d|\varphi|)$ 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 4 may coincide for some levels, until they diverge on the right-hand side of some rule $a \to b_1 \cdots b_k$ 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 $a_t \to b_1 \cdots$, 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 $a'_t \to b_1 \cdots$, 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 2. 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 $\mathcal{O}(g)$ [38], which we show to be the case of L-systems in Section 4. 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 [3] or by balancing the grammar [8]. 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 [35], 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 [13]. Shallit and Swart [36] 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 $\log i$.

3.3. 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 1. *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.

4. Breaking the Repetitiveness Lower Bound δ

The measure δ is a (strict) lower-bound to all the other usually considered repetitiveness measures [18, 24]. It is also a lower bound to the *k-th order empirical entropy*, which is a lower bound for statistical compression [24]. 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 [18], 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.

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

Proof. Kociumaka et al. [18] 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 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 $\tau = \mathtt{id}$.

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 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

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

$s_3 =$ c b a b a a b

$s_6 =$ c b a b a a b a a a b a a a b a a a a a b

Figure 2: All the substrings of length 6 of the string s_6 of Lemma 3 starting inside some position $i \leq |s_3| = 7$ are distinct, because the runs of 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 $s_d = \varphi^d(s)$:

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

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 $s_d = \mathtt{c} \prod_{i=0}^{d-1} \mathtt{a}^i\mathtt{b}$, 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 b's they cross, and the number of a's at their extremes, so they are all distinct. An example can be seen in Figure 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 a's followed by b's, so $\delta = \mathcal{O}(g_{rl}) = \mathcal{O}(\sqrt{n})$. Thereore, it holds $\delta = \Theta(\sqrt{n}) = \Theta(\ell\sqrt{n})$ in this string family. □

The strings of Lemma 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 1 and 3, we obtain that ℓ and δ are uncomparable as repetitiveness measures.

Corollary 1. *The measures ℓ and δ are uncomparable.*

5. Uncomparability of ℓ with other Repetitiveness 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. [18], which needs $\Omega(\log^2 n)$ bits to be represented with any method. This string family will be crucial in the following proofs.

Definition 3 ([18])**.** *The string family $\mathcal{K}$ is formed by all the infinite strings $\boldsymbol{s}$ over $\{\mathtt{a}, \mathtt{b}\}$ constructed as follows:*

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

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 $\ell = \mathcal{O}(g_{rl})$, making L-systems uncomparable to RLCFGs.

Lemma 4. *There exists a string family where $\ell = \Omega(g_{rl} \log n / \log\log n)$.*

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 [18]. 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 $g_{rl} = \mathcal{O}(\log n)$ in this family. Because of this, and because g_{rl} is a reachable measure, it holds that $g_{rl} = \Theta(\log n)$ 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 $\ell = \Omega(g_{rl} \log n / \log\log n)$. □

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 5. *There exists a string family where $\ell = \Omega(z_e \log n / \log\log n)$.*

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 4, 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 a's followed by b, its source is aligned with $\mathtt{a}^n\mathtt{b}$, so $z_e = \Theta(\log n)$. Thus, $\ell = \Omega(z_e \log n / \log\log n)$. □

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

Lemma 6. *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 $\mathcal{O}(\log n)$ b's, its BWT has also $\mathcal{O}(\log n)$ runs of a's separated by b's (or the unique \$). 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 4. *The measure* ℓ *is uncomparable to the repetitiveness measures* δ, γ, b, v, c, g_{rl}, z, z_{no}, z_{end}, z_e, *and* r.

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_e, g_{rl}, or r, which by Lemmas 4, 5, and 6, 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. Macro-systems

In this section we give a first step in combining L-systems with bidirectional macro schemes (BMSs) [37], 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 7 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 4. *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* $w[s_i..s_i + |x_i| - 1] = x_i$*, 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 $w[1\,..\,n]$ 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 5. *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, $\texttt{size}(M) = \sum_{A \in V} |R(A)|$.*

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

Definition 6. *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 $\exp(a) = a$.*
- *If $S \to \varepsilon$, then $\exp(S) = \varepsilon$.*
- *If $A \to B_1 \cdots B_k$ is a rule, then $\exp(A) = \exp(B_1) \cdots \exp(B_k)$.*
- $\exp(A[i : j]) = \exp(A)[i\,..\,j]$.

We say that the macro-system is valid *if there is exactly one solution $w \in \Sigma^*$ for $\exp(S)$. 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 $\exp(S)$ may have infinite solutions, as in $S \to S$ or $S \to S[1 : 2]$. It might also have no solutions, as in $S \to \texttt{a}\,S$ or $S \to S[2 : 3]$. On the other hand, there can be valid solutions involving overlaps, like $S \to \texttt{a}\,S[1 : 3]$, which solves to (only) $\exp(S) = \texttt{aaaa}$.

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 $A \to B^t$ can be translated in macro-systems as $A \to B\, A[1 : (t-1) \cdot |\exp(B)|]$, therefore macro-systems are at least as powerful as collage-systems [16]. The following example shows that they can be asymptotically strictly smaller.

Example 1. *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 $\Theta(\log |F_k|)$ [25, Thm. 32]. Instead, we can mimic a BMS of size 4 [25, Lem. 35] with a constant-sized macro-system generating F_k, as follows (with $f_k = |F_k|$):*

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

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 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 $N = \sum_{A \in V} |\exp(A)|$.*

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

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

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

Once the lengths are calculated, we create strings $E_A[1 \,..\, |\exp(A)|]$ 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 the necessary cells so that E_S has no unknown positions, at which point the decompressed string is $\exp(S) = E_S$.

We successively define the symbols $E_S[1]$ to $E_S[|\exp(S)|]$, which will trigger other definitions. The definition of a symbol $V[r]$ proceeds recursively, as shown in Algorithm 5. Note that we mark the traversed positions with $\perp$ to detect loops that flag the system as invalid. More importantly, although the recursion

Algorithm 5: Defining a symbol when decompressing a macro-system.

Input : Terminal or variable A and position r to define from $\exp(A)$.
Output: Obtains $\exp(A)[r]$ and assigns it to $E_A[r]$, as well as any other position discovered along the process.

```
1  function define(A, r)
2      if r ∉ [1 .. |exp(A)|] then return "invalid system" (out of bounds)
3      if E_A[r] = ⊥ then return "invalid system" (loop detected)
4      if E_A[r] ≠ ? then return E_A[r] (already known)
5      if A = a (a terminal) then return a
6      E_A[r] ← ⊥ (will be defined in the process)
7      if A → B_1 ··· B_k (a variable) then
8          let p be such that Σ_{j=1}^{p-1} |exp(B_j)| < r ≤ Σ_{j=1}^{p} |exp(B_j)|
9          E_A[r] ← define(B_p, r − Σ_{j=1}^{p-1} |exp(B_j)|)
10     else if A = B[i : j] (an extraction) then
11         E_A[r] ← define(B, i + r − 1)
12     return E_A[r]
```

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 5 might suggest that we need a logarithmic-time binary search. We can, instead, precompute arrays $P_A[1\,..\,|\exp(A)|]$ for every $A \to B_1 \cdots B_k$, so that we assign 1 to $P_A[1\,..\,|\exp(B_1)|]$, 2 to $P_A[|\exp(B_1)| + 1\,..\,|\exp(B_1)| + |\exp(B_2)|]$, 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 $k \le |\exp(A)|$ for every A. □

Example 2. *Now we show how to recover the string F_7 from the macro-system with rules*

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

We first determine the expansion lengths of the variables of the macro-system: $|\exp(A)| = 1$, $|\exp(B)| = 1$, and $|\exp(S)| = 13$. 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 5 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* $\texttt{b}$*. 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* $\texttt{b}$ *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 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* $A \to B_1 \cdots B_k \in R$*, 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 $\exp(A)$ 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. *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 $S \to S_1 \cdots S_b$, 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. □

Theorem 7. *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 $|\exp(A)|$ for every variable $A \in R$ as done in the proof of Theorem 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 $\langle S, 1, |\exp(S)|\rangle$. 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

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

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 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 $w[l\,..\,r] = \exp(A)$; we say that $pos(A) = l$ 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 $(a, \bot)$, recording the explicit symbol. If $X_t = A$ is a variable, then the phrase is $(\exp(A), pos(A))$, pointing to the leftmost substring derived from A. Finally, if $X_t = A[i : j]$ is an extraction symbol, then the phrase is $(\exp(A)[i\,..\,j], pos(A) + i - 1)$, also pointing inside the leftmost substring of w derived from A (we detect that the macro-system is invalid if $j > |\exp(A)|$).

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 $b \le m$. □

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 [25].) It is now immediate that we can decompress internal macro-systems in linear time.

Corollary 2. *Given an internal macro-system $M = (V, \Sigma, R, S)$ representing string $w[1\,..\,n]$, we can compute w in time $\mathcal{O}(n)$.*

Proof. The proof of Theorem 7 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, $S_t = S[pos(A) : pos(A) + |\exp(A)| - 1]$ if $X_t = A$ is a variable, and $S_t = S[pos(A) + i - 1 : pos(A) + j - 1]$ if $X_t = A[i : j]$ is an extraction symbol. We can now apply the decompression technique of Theorem 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 $A \to B_1 \cdots B_k$. The total time is then $\mathcal{O}(n)$. □

7. NU-systems

A *NU-system*[2] 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 8. *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, $\Gamma : V \to V$ is a coding, and R is a set of rules where the right-hand sides may contain* extractions*, that is, $R : V \to (V \cup E)^+$ 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 the size of a NU-system is reachable.

7.1. 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 $A \to B_1 \cdots B_k$, then it holds $|A^1| = \sum_{r=1}^{k} |B_r|$, where $|B_r| = 1$ if $B_r \in V$ and

[2] This is an enhanced version of the NU-systems defined in the conference version of this work [26, 27], and should be taken as the definitive one.

$|B_r| = j-i+1$ 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 $B^{l'}[i] \cdot B^{l'}[i+1] \cdots B^{l'}[j]$ 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 8. *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 5), the time is now multiplied by the number of levels used.

Example 3. *Consider the NU-system with rules*

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

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

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

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):

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

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

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

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

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

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

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

With those, we can now complete S^1:

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

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 $T \to S(1)[5:7]\ T(3)[1:3]\ T(1)[7:8]$.

7.2. The measure ν

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

Definition 9. *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 9. *It always holds that $\nu = \mathcal{O}(\min(\ell, m))$.*

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 2, 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 3. *The measures δ and ν are uncomparable.*

Proof. It follows because $\nu = \mathcal{O}(\ell)$ and $\ell = o(\delta)$ on some string families (Lemma 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 10. *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. [18], needing $\Omega(\log^2 m)$ bits to be represented with any method (Def. 3), 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 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 4, 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 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 $\mathtt{1}$. Let i_j be the number of $\mathtt{0}$'s in x between the $(j-1)$-th and the j-th $\mathtt{1}$'s, for $j \in [2,k]$. Also, let i_1 and i_{k+1} be the number of $\mathtt{0}$'s at the left and right extremes of x. We construct the NU-system $U = (V, R, \Gamma, S)$ as follows:

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

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.

7.3. Properties

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

Proposition 2. *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

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

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 $\Gamma'_k : \Sigma'_k \to \Sigma_k$, so that $\Gamma'_k(\langle k, a\rangle) = \Gamma_k(a)$, and then $\Gamma = \Gamma'_1 \cup \Gamma'_2 \cup \{a \to a \mid a \in \Sigma_1 \cup \Sigma_2 \cup \{S\}\}$. It is easy to see that U generates $w_1 \cdot w_2$, and then $\nu(w_1 \cdot w_2) \le \mathtt{size}(U) = \mathcal{O}(\mathtt{size}(U_1) + \mathtt{size}(U_2)) = \mathcal{O}(\nu(w_1) + \nu(w_2))$. □

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

Corollary 4. *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 2, 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 3. *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 5. *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 4 and Proposition 3 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 4. □

Proposition 4. *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. □

8. 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.

Definition 10. *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 $s \to sx$ is a rule, with $x \neq \varepsilon$). We say L is* codingless *if $\tau = \mathtt{id}$.*

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

Definition 11. *The measure $\ell_e(w)$ (resp., $\ell_u(w)$) denotes the size of the smallest expanding (resp., uniform) L-system generating w. The measure $\ell_p(w)$ denotes the size of the smallest prolongable L-system generating w. The measure $\ell_c(w)$ denotes the size of the smallest codingless L-system generating w. The measure $\ell_{pc}(w)$ denotes the size of the smallest prolongable and codingless L-system generating w. The measure $\ell_{pu}(w)$ 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 [30]. Some of these classes also differ in the factor complexity of the sequences they can generate [31]. 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 [1]. 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 $\mathcal{O}(\ell_p)$ space), Algorithm 6 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 $\mathcal{O}(\ell_e)$ 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 1. The cost for extracting $w[i\,..\,j]$ we obtained in Section 3.2 is also reduced, to $\mathcal{O}(|\varphi| \log n)$ preprocessing space and time, and $\mathcal{O}(j - i + \log n)$ extraction time. Further, if a system is k-uniform (i.e., using $\mathcal{O}(\ell_u)$ space), then we know easily the size to which every symbol expands after l levels, in which case we can efficiently extract $w[i\,..\,j]$ without need of any preprocessing or extra space: in lines 6–7 of Algorithm 4 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 ℓ_e (and ℓ_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.

Algorithm 6: 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.

```
1  function decompress(s, n)
2      w[1] ← s
3      r ← 1
4      p ← 1
5      while true do
6          let w[r] → b_1 ··· b_k ∈ φ
7          r ← r + 1
8          for i ← 1 to k do
9              w[p] ← b_i
10             output τ(b_i)
11             if p = n then return
12             p ← p + 1
```

Lemma 7. *For any L-system L, there exists a CFG G of size $|G| = \mathcal{O}(d \cdot \texttt{size}(L))$ generating the same string. Further, it always holds that $g = \mathcal{O}(\ell_e \log n)$.*

Proof. Let $L = (\Sigma, \varphi, \tau, s, d, n)$ be an L-system generating $w[1\,..\,n]$. 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 $i \in [0\,..\,d-2]$, 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 $A \to B_1 \cdots B_k$ appearing at depth $0 \le i \le d-2$ of the L-system, we add $A_i \to (B_1)_{i+1} \cdots (B_k)_{i+1}$ to the set of rules R. Further, for each rule $A \to B_1 \cdots B_k$ appearing at depth $d-1$ in L, we add the grammar rule $A_{d-1} \to \tau(B_1) \cdots \tau(B_k)$ 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 $(d+1) \cdot \texttt{size}(L)$ and generates a string w^+, of which the desired string $w[1\,..\,n]$ is a prefix.

We now modify G to generate exactly $w[1\,..\,n]$. 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 $A_i \to (B_1)_{i+1} \cdots (B_k)_{i+1}$ whose expansion must be pruned to length l_i, we define k_i as the maximum position $j < k$ such that $|\exp((B_1)_{i+1} \cdots (B_j)_{i+1})| <$

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 $l_{i+1} = l_i - |\exp((B_1)_{i+1} \cdots (B_{k_i})_{i+1})|$ of $(B_{k_i+1})_{i+1}$. We therefore create a new rule $A'_i \to (B_1)_{i+1} \cdots (B_{k_i})_{i+1} \cdot (B_{k_i+1})'_{i+1}$, 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 $w[1\,..\,n]$ 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 $\mathcal{O}(\mathtt{size}(L) \log n)$. Thus, $g = \mathcal{O}(\ell_e \log n)$. □

As $\ell_c = \mathcal{O}(g)$ by Lemma 2 (the coding used in the proof has $\tau = \mathtt{id}$), and $g = \mathcal{O}(\ell_e \log n)$ by Lemma 7, we obtain the following corollary.

Corollary 6. *It always holds that* $\ell_c = \mathcal{O}(\ell_e \log n)$.

Surprisingly, all the restricted L-systems (except possibly uniform systems) outperform δ on some string family. We already showed this for ℓ_{pc} (and thus ℓ_p and ℓ_c) in Lemma 3, where the L-system we used was prolongable and without coding. The next lemma proves that the same holds for ℓ_e.

Lemma 8. *There exists a string family where* $\ell_e = \mathcal{O}(1)$ *and* $\delta = \Omega(\log n)$.

Proof. We use a small modification of the D0L-sequence described by Ehrenfeucht et al. [6, 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 $\ell_e = \mathcal{O}(1)$ 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 ℓ_p. That is, restricting L-systems to be prolongable yields a weaker measure. We will actually prove that ℓ_p can be asymptotically larger than ℓ_c, the L-systems without codings.

Lemma 9. *There exists a string family where* $\ell_p = \Omega(\ell_c \log n / \log\log n)$.

Proof. Let $\mathcal{F} = \{\mathtt{a}^{n-1}\mathtt{b} \,|\, n \geq 1\}$. Clearly, ℓ_c is constant in this string family: the L-system $L_n = (\Sigma, \varphi, \tau, s, d, n)$ where $\Sigma = \{\mathtt{a}, \mathtt{b}\}, \varphi = \{\mathtt{a} \to \mathtt{a}, \mathtt{b} \to \mathtt{ab}\}, \tau = \mathtt{id}, s = \mathtt{b}$, 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 $\mathtt{a}^{n-1}\mathtt{b}$. Let $k = |\Sigma_n|$ and $t = \mathtt{width}(\varphi_n) > 1$. Observe that it is only necessary to have one symbol $c \in \Sigma_n$ with $\tau_n(c) = \mathtt{b}$ because there is only one $\mathtt{b}$ in $\mathtt{a}^{n-1}\mathtt{b}$, so w.l.o.g. assume that $\mathtt{b} \in \Sigma_n$ and $\tau_n(\mathtt{b}) = \mathtt{b}$. As the system is prolongable, each level is a prefix of the next one. This implies that the morphism should be iterated until $\mathtt{b}$ 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, $\mathtt{b}$ 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 $\mathtt{b}$ 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, $\ell_p \geq k \geq \log_t n$ and $\ell_p \geq t$, so $\ell_p \geq \max(t, \log_t n)$. 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, $\ell_p = \Omega(\ell_c \log n / \log\log n)$ in this string family. □

We can prove a similar result for uniform systems.

Lemma 10. *There exists a string family where* $\ell_p = \Omega(\ell_u \log n / \log\log n)$.

Proof. It is not difficult to see that ℓ_u is constant in the family $\{\mathtt{a}^{2^k}\mathtt{b} \,|\, k \geq 0\}$: consider the axiom $s = \mathtt{c}$ and the rules $\mathtt{c} \to \mathtt{ab}$, $\mathtt{a} \to \mathtt{aa}$, $\mathtt{b} \to \mathtt{bb}$, the level $d = k$

and the prefix length $n = 2^k + 1$. The same argument as in Lemma 9 yields that $\ell_p = \Omega(\ell_u \log n / \log\log n)$ for this string family. □

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

Lemma 11. *There exists a string family where* $\ell_{pc} = \Omega(\ell_p\sqrt{n})$.

Proof. We prove that $\ell_{pc} = \Theta(n)$ on $\mathcal{F} = \{\mathtt{0}^{n-1}\mathtt{1} \mid n \geq 2\}$, whereas $\ell_p = \mathcal{O}(\sqrt{n})$. Any minimal codingless prolongable L-system generating $\mathtt{0}^{n-1}\mathtt{1}$ must contain the rule $\mathtt{0} \to \mathtt{0}^{n-1}\mathtt{1}$, which implies $\ell_{pc} = \Theta(n)$. 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 $\mathtt{0}$s and $\mathtt{1}$s, otherwise undesired symbols would appear in the final string. If x does not contain $\mathtt{1}$s, then $\mathtt{1}$ is unreachable from $\mathtt{0}$, which is a contradiction. So, it must be the case that x contains a least one $\mathtt{1}$, 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

$$\begin{aligned}
&\varphi^0(\mathtt{a}) = \mathtt{a}\\
&\varphi^1(\mathtt{a}) = \mathtt{ab}\\
&\varphi^2(\mathtt{a}) = \mathtt{abc}^{k-1}\mathtt{d}\\
&\varphi^3(\mathtt{a}) = \mathtt{abc}^{k-1}\mathtt{d0}^{(\lfloor\sqrt{n-1}\rfloor-1)(k-1)}\mathtt{0}^{\lfloor\sqrt{n-1}\rfloor-3+j}\mathtt{1},
\end{aligned}$$

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 $\mathtt{1}$ is mapped to $\mathtt{1}$ 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 7. *There exists a string family where* $\ell_{pc} = \Omega(\ell_c n)$.

Proof. Just note that ℓ_c is constant in the family used in Lemma 11. □

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

Corollary 8. *The measure* ℓ_{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 5. *There exists a string family where* $\ell_e = \Omega(\ell_{pc}\sqrt{n}/\log n)$.

Proof. Such a family is the one of Lemma 3. In this family, $\ell_{pc} = \mathcal{O}(1)$. On the other hand, from Lemma 7, it holds that $\ell_e = \Omega(g/\log n) \subseteq \Omega(\delta/\log n)$. 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, $\ell_e = \Omega(\sqrt{n}/\log n) = \Omega(\ell_{pc}\sqrt{n}/\log n)$. □

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

Lemma 12. *There exists a string family where* $\ell = o(\ell_c)$.

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 $0 \leq i \leq d'$. Then, since α' is a constant, there must exist a period of length q' starting at $b'_{p'}$, with $p' + q' \leq \alpha' \leq d'$, and the corresponding values $t' = d' - j'q'$ and $p' \leq t' < p' + q'$, 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 $r \geq 1$. 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 [35], 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 [35, 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 $\varphi(c_i) = x_{i+1}c_{i+1}y_{i+1}$

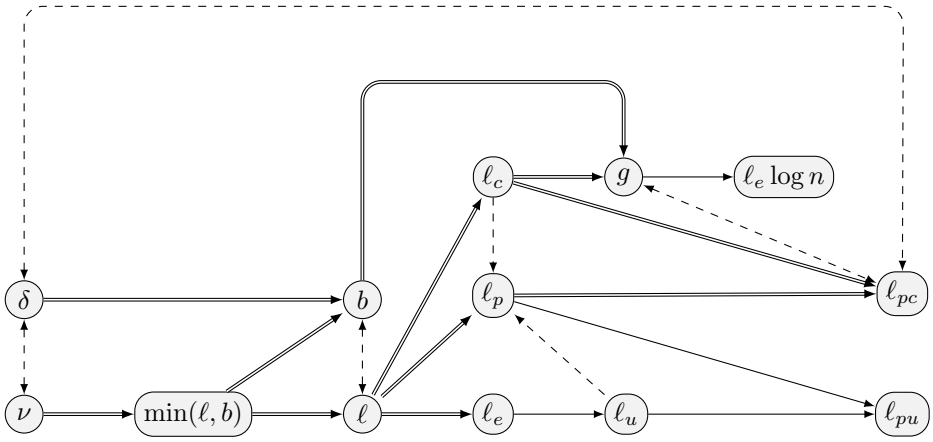

Figure 3: Asymptotic relations between the measure ℓ and its variants, the measure ν, and other relevant state-of-the-art repetitiveness measures. A solid arrow from a measure v_1 to a measure v_2 means that it always holds that $v_1 = \mathcal{O}(v_2)$. A double solid arrow from v_1 to v_2 means that 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 $v_1 = o(v_2)$.

(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 $\texttt{width}(\varphi)$ and no expanding symbols. Hence, φ grows polynomially on them. On the other hand, c_d is an expanding symbol, hence distinct from `a` and `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 3, which also includes the measure ν from Section 6.

9. Conclusions and Further Work

We have contributed to the study of string repetitiveness, by introducing and studying repetitiveness measures that are related to string morphisms. Our first measure, the size ℓ of the smallest L-system that generates the string, captures in a sense structured repetitiveness, as introduced by the repeated application of a string morphism. Surprisingly, while on some string families ℓ can be (slightly) larger than various copy-paste-based measures of repetitiveness, like the size of Lempel-Ziv parses or the number of runs in the Burrows-Wheeler Transform, in others it can outperform them all by a large margin, $\Theta(\sqrt{n})$. This suggests

that the structured repetitiveness captured by ℓ is out of the reach of current measures. On the other hand, ℓ is upper bounded by the size of any context-free grammar that generates the string, so it is never excessively larger than copy-paste methods (which are all related by a polylogarithmic factor). We also show that L-systems allow efficient access to random positions of the string they represent, and that several simplifications or restrictions of L-systems lead to weaker measures. We believe that computing the size ℓ of the smallest L-system is NP-hard, but the question is left open, as well as approximating ℓ. Another question is whether we can build string indices of size $\mathcal{O}(\ell \operatorname{polylog} n)$. We also leave open other questions on L-systems, like whether $\ell_u = \Omega(\delta)$ (where ℓ_u is the size of the smallest uniform L-system) and whether g can be $o(\ell_e \log n)$ in some string family (where ℓ_e is the size of the smallest expanding L-system).

We also combined L-systems with the best reachable copy-paste mechanism, the bidirectional macro schemes, into a new formalism we call NU-systems, whose minimum size ν is a new measure of repetitiveness that strictly outperforms both. While NU-systems are possibly too powerful to enable efficient random access, they can still be decompressed with reasonable efficiency (that is, related to the set of strings it generates, one of which is the output string).

There are several open questions related to NU-systems and ν, starting from whether computing ν is NP-hard and whether it can be approximated. Another question is whether we can decompress NU-systems in time related to the length of the decompressed string. Other questions are whether $\nu = \Omega(\ell \log\log n / \log n)$, or $\nu = \Omega(\delta/\sqrt{n})$, for every string family, and whether is ν is $\mathcal{O}(\gamma)$, or at least $o(\gamma \log(n/\gamma))$, for every string family (recall that $o(\gamma \log(n/\gamma))$ space is unknown to be reachable [15]).

In a more general perspective, this paper pushes a little further the discussion of what we understand by repetitiveness. Intuitively, repetitiveness is about copies, and bidirectional macro schemes arguably capture those copies as much as possible, but there are other repetitions that can be captured better than by explicit copies. L-systems capture a kind of structured copies, and NU-systems capture those structured and arbitrary copies. What other regularities could we exploit when compressing strings, keeping the representation simple to handle and the associated repetitiveness measure (hopefully efficiently) computable?

Acknowledgments

Both authors were funded with Basal Funds FB0001 and AFB240001, and Fondecyt Grant 1-230755, ANID, Chile. C.U. was also funded by scholarship ANID-Subdirección de Capital Humano/Doctorado Nacional/2021-21210580, ANID, Chile.

References

[1] Allouche, J., Shallit, J.O.: Automatic Sequences - Theory, Applications, Generalizations. Cambridge University Press (2003)

[2] Bannai, H., Funakoshi, M., I, T., Köppl, D., Mieno, T., Nishimoto, T.: A separation of γ and b via Thue–Morse words. In: Proc. 28th International Symposium on String Processing and Information Retrieval (SPIRE). pp. 167–178 (2021)

[3] Bille, P., Landau, G.M., Raman, R., Sadakane, K., Rao, S.S., Weimann, O.: Random access to grammar-compressed strings and trees. SIAM Journal on Computing **44**(3), 513–539 (2015)

[4] Burrows, M., Wheeler, D.: A block sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equipment Corporation (1994)

[5] Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Sahai, A., Shelat, A.: The smallest grammar problem. IEEE Transactions on Information Theory **51**(7), 2554–2576 (2005)

[6] Ehrenfeucht, A., Lee, K., Rozenberg, G.: Subword complexities of various classes of deterministic developmental languages without interactions. Theoretical Computer Science **1**(1), 59–75 (1975)

[7] Gallant, J.K.: String Compression Algorithms. Ph.D. thesis, Princeton University (1982)

[8] Ganardi, M., Jeż, A., Lohrey, M.: Balancing straight-line programs. Journal of the ACM **68**(4), article 27 (2021)

[9] Gasieniec, L., Karpinski, M., Plandowski, W., Rytter, W.: Efficient algorithms for Lempel-Ziv encoding. In: Proc. 5th Scandinavian Workshop on Algorithm Theory (SWAT). pp. 392–403 (1996)

[10] Gasieniec, L., Kolpakov, R., Potapov, I., Sant, P.: Real-time traversal in grammar-based compressed files. In: Proc. 15th Data Compression Conference (DCC). p. 458 (2005)

[11] Giuliani, S., Inenaga, S., Lipták, Z., Prezza, N., Sciortino, M., Toffanello, A.: Novel results on the number of runs of the Burrows-Wheeler-Transform. In: Proc. Theory and Practice of Computer Science (SOFSEM). pp. 249–262 (2021)

[12] Jeż, A.: Approximation of grammar-based compression via recompression. Theoretical Computer Science **592**, 115–134 (2015)

[13] Kaltofen, E., Villard, G.: On the complexity of computing determinants. Computational Complexity **13**, 91–130 (2004)

[14] Kempa, D., Kociumaka, T.: Resolution of the Burrows-Wheeler Transform conjecture. Communications of the ACM **65**(6), 91–98 (2022)

[15] Kempa, D., Prezza, N.: At the roots of dictionary compression: String attractors. In: Proc. 50th Annual ACM SIGACT Symposium on Theory of Computing (STOC). p. 827–840 (2018)

[16] Kida, T., Matsumoto, T., Shibata, Y., Takeda, M., Shinohara, A., Arikawa, S.: Collage system: a unifying framework for compressed pattern matching. Theoretical Computer Science **298**(1), 253–272 (2003)

[17] Kociumaka, T., Navarro, G., Olivares, F.: Near-optimal search time in δ-optimal space, and vice versa. Algorithmica **86**, 1031–1056 (2024)

[18] Kociumaka, T., Navarro, G., Prezza, N.: Towards a definitive compressibility measure for repetitive sequences. IEEE Transactions on Information Theory **69**(4), 2074–2092 (2023)

[19] Kreft, S., Navarro, G.: LZ77-like compression with fast random access. In: 2010 Data Compression Conference (DCC). pp. 239–248 (2010)

[20] Lempel, A., Ziv, J.: On the complexity of finite sequences. IEEE Transactions on Information Theory **22**(1), 75–81 (1976)

[21] Lindenmayer, A.: Mathematical models for cellular interactions in development I. Filaments with one-sided inputs. Journal of Theoretical Biology **18**(3), 280–299 (1968)

[22] Lindenmayer, A.: 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)

[23] Munro, J.I., Raman, R., Raman, V., Rao, S.S.: Succinct representations of permutations and functions. Theoretical Computer Science **438**, 74–88 (2012)

[24] Navarro, G.: Indexing highly repetitive string collections, part I: Repetitiveness measures. ACM Computing Surveys **54**(2), article 29 (2021)

[25] Navarro, G., Ochoa, C., Prezza, N.: On the approximation ratio of ordered parsings. IEEE Transactions on Information Theory **67**(2), 1008–1026 (2021)

[26] Navarro, G., Urbina, C.: On stricter reachable repetitiveness measures. In: Proc. 28th International Symposium on String Processing and Information Retrieval (SPIRE). pp. 193–206 (2021)

[27] Navarro, G., Urbina, C.: L-systems for measuring repetitiveness. In: 34th Annual Symposium on Combinatorial Pattern Matching, CPM 2023. pp. 25:1–25:17 (2023)

[28] Navarro, G., Urbina, C.: Iterated straight-line programs. In: Proc. 16th Latin American Symposium on Theoretical Informatics (LATIN), part I. pp. 66–80. LNCS 14578 (2024)

[29] Nishimoto, T., I, T., Inenaga, S., Bannai, H., Takeda, M.: Fully dynamic data structure for LCE queries in compressed space. In: 41st International Symposium on Mathematical Foundations of Computer Science (MFCS). pp. 72:1–72:15 (2016)

[30] Pansiot, J.: On various classes of infinite words obtained by iterated mappings. In: Automata on Infinite Words. Lecture Notes in Computer Science, vol. 192, pp. 188–197 (1984)

[31] Pansiot, J.: Subword complexities and iteration. Bulletin of the European Association for Theoretical Computer Science **26**, 55–62 (1985)

[32] Przeworski, M., Hudson, R., Di Rienzo, A.: Adjusting the focus on human variation. Trends in Genetics **16**(7), 296—302 (2000)

[33] Rozenberg, G., Salomaa, A.: The Mathematical Theory of L Systems, pp. 161–206. Springer US, Boston, MA (1976)

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

[35] Salomaa, A.: On exponential growth in Lindenmayer systems. Indagationes Mathematicae (Proceedings) **76**(1), 23–30 (1973)

[36] Shallit, J., Swart, D.: An efficient algorithm for computing the ith letter of $\varphi^n(a)$. In: Proc. 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 768–775 (1999)

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

[38] Verbin, E., Yu, W.: Data structure lower bounds on random access to grammar-compressed strings. In: Proc. 24th Annual Symposium on Combinatorial Pattern Matching (CPM). pp. 247–258 (2013)