FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

Assignment of master's thesis

Title:	Algorithms for Combinatorics on Words
Student:	Martin Rejmon
Supervisor:	doc. Ing. Štěpán Starosta, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2021/2022

Instructions

1) Get acquainted with the computer algebra system SageMath and its up-to-date development standards.
2) Survey the library available in SageMath related to combinatorics on words (finite and infinite words, morphisms). Perform a time and space complexity analysis of selected non-trivial algorithms in the library.
3) Design and implement changes and new functions/methods in the library. Focus on algorithms for detection of morphisms which are "pushy", "unboundedly repetitive", injective, and injective on language; focus on testing whether a factor is a factor of the given language generated by a morphism; or other convenient non-trivial additions.

4) Implement the changes in accordance with SageMath developer standards, and start the integration process of your changes of at least one non-trivial change/addition.

_

K. Klouda, Š. Starosta, An Algorithm Enumerating All Infinite Repetitions in a D0L-System, Journal of Discrete Algorithms 33 (2015), 130–138, DOI: 10.1016/j.jda.2015.03.006
K. Klouda, Š. Starosta, Characterization of circular D0L-systems, Theoretical Computer Science (2019), 131-137, DOI:10.1016/j.tcs.2019.04.021
K. Klouda: Bispecial factors in circular non-pushy D0L languages, Theoretical Computer Science, 445 (2012), 63-74; DOI: 10.1016/j.tcs.2012.05.007

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 28 October 2020 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Algorithms for Combinatorics on Words

Bc. Martin Rejmon

Department of Theoretical Computer Science

Supervisor: doc. Ing. Štěpán Starosta, Ph.D.

May 6, 2021

Acknowledgements

I would like to thank my supervisor doc. Ing. Štěpán Starosta, Ph.D. for his guidance and patience.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on May 6, 2021 . . . . . .

Czech Technical University in Prague

Faculty of Information Technology

Citation of this thesis

Rejmon, Martin. *Algorithms for Combinatorics on Words*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

V této práci je podrobně popsáno několik algoritmů řešících problémy z matematického oboru kombinatorika na slovech. Součástí práce je i jejich implementace v svobodném a otevřeném počítačovém algebraickém systému SageMath za účelem jejich integrace do téhož systému. Mezi zkoumané algoritmy patří: klasifikace rostoucích písmenek morfismu, rozhodování se, zda morfismus je prostý, hledání zjednodušení morfismu, hledání všech nekonečných opakování v D0L systému a hledání všech podřetězců (kratší než zadaná délka) slov jazyka PD0L systému. Dále je v práci diskutován problém rozhodování se, zda morfimus D0L systému je prostý na množině podřetězců slov jazyka toho systému. Není obecně známo, zda tento problém jde rozhodnout, což v této práci není vyřešeno, ale je zde uvedeno, proč je to netriviální problém a kde některé z možných způsobů jeho řešení selžou.

Klíčová slova rostoucí písmenka morfismu, prosté morfismy, opakující se D0L systémy, morfismy prosté na jazyku

Abstract

In this thesis, several algorithms for problems from the mathematical field combinatorics on words are thoroughly explained. The algorithms are also implemented in the free and open-source mathematics software system SageMath with the goal of their eventual integration into the same system. The algorithms include: classifying mortal and bounded letters of a morphism, deciding whether a morphism is injective, finding a simplification of a morphism, finding all infinite repetitions in a D0L system, and finding all factors (up to a certain length) of words of the language of a PD0L system. Furthermore, the problem of deciding whether a morphism of a D0L system is injective on the set of factors of words of the language of the system is discussed. The decidability of this problem is an open question, which is not answered in this thesis, but it is shown why it is nontrivial and where some approaches to solving this problem fail.

Keywords growing letters of a morphism, injective morphisms, repetitive D0L systems, morphisms injective on language

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

Combinatorics on words straddles the border of discrete mathematics and theoretical computer science. It studies the combinatorial properties of words – finite or infinite sequences of symbols (letters) from some set (alphabet). Here is an example of a simple problem from combinatorics on words: If we have two finite words x and y such that $x \times y = y \times x$ (where $\times$ denotes the (non-commutative) operation of concatenation), then there must be another word z and two positive integers i and j such that $x = z^i$ and $y = z^j$ [1] (Proposition 1.3.2).

This thesis focuses on selected properties of Lindenmayer systems (abbreviated as L-systems), described by Aristid Lindenmayer in his paper published in 1968 [2]. Loosely speaking, they could be seen as a counterpart to formal grammars from formal language theory, with two notable differences:

- The rewriting rules are all carried out *in parallel*, instead of sequentially.
- There is no distinction between terminal and nonterminal symbols.

On the one hand, the first difference gives L-systems interesting generative capabilities. For example, it can be shown that the language $\{a^{2^i} \mid i \geq 0\}$ is context-sensitive but is not context-free, however, it can be easily generated with the deterministic context-free L-system defined by the rewriting rule $a \mapsto aa$ and the starting word a.

Moreover, we can study with deterministic L-systems not only the languages (sets of words) they generate but also the sequences of words they generate. For example, take the deterministic context-free L-system defined by the rewriting rules $a \mapsto b$ and $b \mapsto ab$ and the starting word b. The first few

words in the sequence of words it generates are: b, a, ab, bab, $abbab$, $babab bab$, ... It can be shown that the lengths of the words in this sequence form the well-known Fibonacci sequence: 1, 1, 2, 3, 5, 8, ...

On the other hand, the second difference significantly restricts the generative power of L-systems. For example, it can be shown that there exists no context-free L-system generating the finite language $\{a, aaa\}$. The reasoning for having only a single alphabet for both terminals and nonterminals is that L-systems were not conceived by A. Lindenmayer as a parallel counterpart to formal grammars, but as a mathematical formalism for the description of simple multicelullar organisms in biology, where the rewriting rules represent cell division, which can happen in multiple cells of an organism at once. As such, it does not make much sense to differentiate between terminal and nonterminal cells.

However, the lack of terminal letters makes it difficult to compare the generative power of parallel rewriting and sequential rewriting. That is why, as L-systems started to be picked up by formal language theorists, numerous extensions regarding terminal letters were devised. One of these extensions will be described much later in this thesis, however, the vast majority of the time only nonextended deterministic context-free systems (abbreviated as D0L systems) will be discussed.

D0L systems are notable in the context of combinatorics on words, as their rewriting rules can be described using (endo)morphisms on words. It should be noted that most areas of combinatorics on words are closely related to various concepts in abstract algebra [3]. However, these connections will not be explored much in this thesis. What *will* be explored in this thesis are the following two problems: deciding whether the language generated by a D0L system is repetitive and deciding whether a D0L system is injective.

A language is *repetitive* if for each positive integer k there is some word in the language, such that it has a factor (contiguous subsequence), which is a k-power (= is of the form v^k for some nonempty word v). The problem of deciding this was studied, for example, by Andrzej Ehrenfeucht and Grzegorz Rozenberg in their paper published in 1983 [4], by Yuji Kobayashi and Friedrich Otto in their paper published in 2000 [5] and by Karel Klouda and Štěpán Starosta in their paper published in 2015 [6].

The algorithm introduced by K. Klouda and Š. Starosta is described in this thesis in great detail [1]. Furthermore, the main output of this thesis (other

[1] It should be noted, that deciding a D0L language's repetitiveness is only a secondary purpose of this algorithm, as will be explained later.

than this text) is an implementation of this algorithm using the free and open-source mathematics software system SageMath [7] (and its consequential integration into the same system). The implementation contains not only the main algorithm itself but also a significant number of various other nontrivial algorithms, upon which the main algorithm depends. Each of these supplemental algorithms is also described in this thesis, together with detailed pseudocode. The list is as follows:

- Algorithms for classifying mortal and bounded letters of a morphism. These are slightly improved versions of the algorithms by Y. Kobayashi and F. Otto [5].
- Algorithm for deciding whether a morphism is injective. This is the algorithm informally described by R. G. Gallager [8] and formally by S. Even [9] for the equivalent problem of deciding whether a code is uniquely decodable.
- Algorithm for finding a simplification of a morphism. This algorithm is patched together from the works of A. Ehrenfeucht and G. Rozenberg [10], T. Harju and J. Karhumäki [11] , and Y. Kobayashi and F. Otto [5].

An emphasis was placed on analysing time and memory complexity, and as a result, a tighter bound on time complexity was obtained (from polynomial to linear) for the problems of classifying mortal and bounded letters, and for the problem of deciding whether a D0L system is pushy.

Moving on to the second studied problem. A D0L system is *injective* if all words, which are a factor of some word of the language of the system, are distinct from each other when they are rewritten once by the system. This subclass of D0L systems was mentioned, for example, by Julien Cassaigne in his paper published in 1994 [12] and by K. Klouda and Š. Starosta in their paper published in 2019 [13]. It is not known whether this problem is decidable. This thesis explains why this problem is nontrivial by showing where two ways of tackling this problem fall short. The first one generates all the factors of words of the language of the system up to a certain length, and the second one tries to approach it as a problem from formal language theory. In particular, it is shown that:

- There exists a sequence of D0L systems that grows linearly in size, but the length of the smallest pair of distinct factors, which are the same when rewritten by the system, grows exponentially.

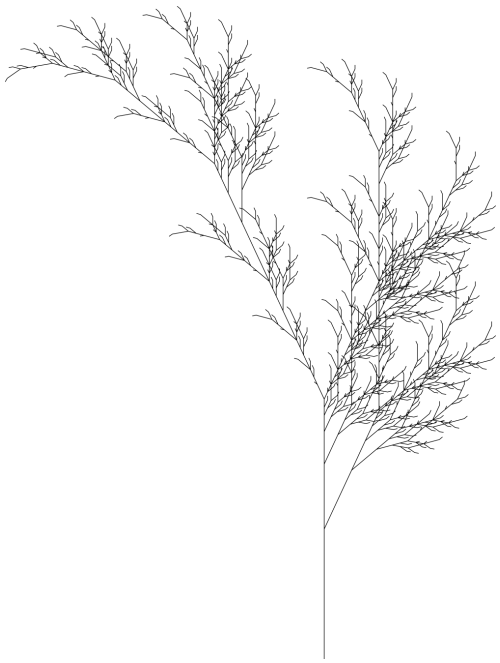

Figure 1.1: Fractal plant-like structure generated using an L-system. See Appendix B for more information.

- The language of pairs of distinct words (but not necessarily factors) arranged in a particular fashion, which are the same when rewritten by the system, is regular.
- The set of all factors of all words in an E0L language is an E0L language.

The motivation for studying these two problems is their connection to the concepts of factor complexity and pattern avoidance, which are prominent concepts in combinatorics on words [14] [12].

Finally, while it is not entirely related to this thesis, it would be amiss to introduce L-systems and not to mention their popular application in generating graphical images depicting various fractal and/or natural (mainly plant-like) patterns. An example can be seen in Figure 1.1.

In Chapter 1, some basic definitions are mentioned. In Chapter 2, K. Klouda's and Š. Starosta's algorithm for deciding the repetitiveness of a language generated by a D0L system is explained, together with the various auxiliary algorithms. In Chapter 3, the implementation of these algorithms is described in the context of SageMath's combinatorics on words module. In Chapter 4, the problem of deciding whether a D0L system is injective is discussed.

Chapter 1

Preliminaries

The definitions in the first two sections can be found, for example, in the book Algebraic Combinatorics on Words [3] by Monsieur Lothaire. The definitions in the third section can be found, for example, in the book The Mathematical Theory of L Systems [15] by G. Rozenberg and Arto Salomaa. The definitions in the fourth section can be found, for example, in the book Introduction to Automata Theory, Languages and Computation by John Edward Hopcroft and Jeffrey David Ullman [16]. The definitions in the fifth section can be found, for example, in the book Graph Theory by Reinhard Diestel [17].

1.1 Finite words

An *alphabet* A is a finite set, whose elements are called *letters*. A *word* w is a finite sequence of letters (indexed from 1). The length of a word w is denoted by $|w|$. The empty word is denoted by ε. The set of all words over an alphabet A is denoted by A^* and the set of all nonempty words over an alphabet A is denoted by A^+.

As usual, commas delineating individual letters in words are omitted (that is, $u = u_1u_2u_3$ instead of $u = u_1, u_2, u_3$) and letters and single-letter words are treated more or less interchangeably. Similarly, the operation of concatenation is denoted by juxtaposition instead of a symbol (that is, $w = uv$ instead of $w = u \times v$).

A word v is a *factor* of a word w, if $w = xvy$ for some words x and y. If x is empty, then v is a *prefix* of w. Similarly, if y is empty, then v is a *suffix* of w. A factor/prefix/suffix v of a word w is called *proper* if $|v| < |w|$. A word u is a *conjugate* of a word v if $u = xy$ and $v = yx$ for some words x and y, that is, u is a "cyclic rotation" of v. For example, $u = aaba$ is a conjugate of

$v = aaab$, where $x = aab$ and $y = a$.

A word w is a *k-power* for some integer $k \geq 0$ if there exists a word v such that w is equal to v repeated k times, this is denoted by $w = v^k$, and the word v is called the *primitive root* of w. Specifically, $x^0 = \varepsilon$ for any word x. If a word w is a k-power for only $k = 1$, then w is *primitive*, otherwise it is *periodic*. A 2-power is called a *square*. Similarly, a 3-power is called a *cube*.

1.2 Infinite words

An *infinite word* is an infinite sequence of letters. Before moving on to additional definitions, let me give a nontrivial example of an infinite word defined on the binary alphabet $\{0, 1\}$ by the following informal algorithm:

1. Start with 0.
2. Take what you already have, flip 0s and 1s, and append this to what you already have.
3. Repeat the previous step ad infinitum.

Adhering to these instructions, we get the following sequence of words:

$$\begin{aligned} t_0 &= 0 \\ t_1 &= 01 \\ t_2 &= 0110 \\ t_3 &= 01101001 \\ t_4 &= 0110100110010110 \\ &\vdots \end{aligned}$$

The limit $\lim_{n\to\infty} t_n$ is equal to the well-known Thue-Morse word. It has several interesting properties (for example none of its factors are cubes) and crops up in a few unrelated places in mathematics [18].

There are several general "frameworks" used for defining infinite words. A simple one is obtained by naturally expanding the concept of a k-power to infinity. An infinite word w is *periodic* if $w = vvv\cdots = \lim_{n\to\infty} v^n = v^\omega$ for some word v, otherwise it is *aperiodic*. However, this does not help us to define, for example, the Thue-Morse word, as it is aperiodic.

A better framework can be obtained by using morphisms. Let A and B be alphabets. A mapping φ from A^* to B^* is a *morphism* if for all words a in A^* and all words b in B^*, we have $\varphi(xy) = \varphi(x)\varphi(y)$.

It can be shown that we can fully define morphisms just by defining the images for all the letters of A (the operation of defining an image y for a letter x is denoted by $x \mapsto y$). The image of a word from A^* is then obtained by concatenation. For example, let $\psi\colon \{1,2,3\}^* \to \{a,b,c\}^*$ be a morphism defined by the images $1 \mapsto ab$, $2 \mapsto c$, and $3 \mapsto \varepsilon$. Then $\psi(1231) = abcab$. As the alphabets can be figured out from the context, I will refrain from explicitly mentioning them and the morphisms will be denoted by grouping images of letters inside braces (such as $\psi = \{1 \mapsto ab, 2 \mapsto c, 3 \mapsto \varepsilon\}$ for the above example).

Usually, we want B to be equal to (or at least a subset of) A, as this allows us to use φ on some word w multiple times in a row, denoted by $\varphi^k(w)$ for k repeats, with $\varphi^0(w) = w$. This will also be assumed in the most of this thesis.

Now, a letter a is *prolongable* on a morphism φ, if $\varphi(a) = av$ for some nonempty word v. Therefore, by repeatedly applying φ to a, we get:

$$\begin{aligned}
\varphi^0(a) &= a \\
\varphi^1(a) &= av \\
\varphi^2(a) &= av\varphi(v) \\
\varphi^3(a) &= av\varphi(v)\varphi^2(v) \\
\varphi^3(a) &= av\varphi(v)\varphi^2(v)\varphi^3(v) \\
&\vdots \\
\varphi^\omega(a) &= \lim_{n\to\infty} \varphi^n = av\varphi(v)\varphi^2(v)\varphi^3(v)\dots
\end{aligned}$$

If[2] $|\varphi^\omega(v)| > 0$, then $w = \varphi^\omega(a)$ defines an infinite word. Such words are called *purely morphic*. Furthermore, a word p is called a *fixed point* of a morphism φ if $\varphi(p) = p$ (a word whose image is itself). It follows that all purely morphic words are fixed points of their respective morphisms (and there cannot be another fixed point starting with the same letter).

Putting all this together, let $T = \{0 \mapsto 01, 1 \mapsto 10\}$ be a morphism. When we repeatedly apply it to 0, we get the sequence of words depicted above (1.2) and the Thue-Morse word is equal to $T^\omega(0)$, therefore it is a purely morphic word and a fixed point of T starting at 0.

Let me use two shorthands to simplify further wording: by letters of a morphism I mean the letters from the alphabet of its domain, and by images of a morphism I mean the images of its letters. Finally, a morphism is *erasing* if it has at least one empty image. These kinds of morphisms are notable, as they usually cause quite a bit of trouble when trying to prove things for an arbitrary morphism.

[2] That is, if v is immortal, as will be defined later.

Lastly, the composition of morphisms φ and ψ is denoted by $\psi \circ \varphi$ and it carries its usual meaning.

1.3 D0L systems

While more a part of formal language theory than combinatorics on words, D0L systems are closely linked to purely morphic words, as they can be seen as their generalization.

A *deterministic context-free Lindenmayer system* (abbreviated as a *D0L system*, where the second letter is a zero, not O) G is a triplet (A, φ, s), where A is an alphabet, $\varphi\colon A^* \to A^*$ is a morphism and s is a word from A^*, also called the *axiom* of G. A *language* is a set of words. The *language* of G is the set $\{\varphi^n(s) \mid n \geq 0\}$, denoted by $L(G)$. Due to the way the language is created, we can similarly define the *sequence* of G as the sequence $(\varphi^n(s))_{n\geq 0}$, denoted by $S(G)$. We also have that a language L is a *D0L language* if there exists some D0L system G such that $L = L(G)$.

Since the alphabet can again be inferred from the context, I will denote D0L systems with just (φ, s). For example, let $H = (\{a \mapsto c, b \mapsto cda, c \mapsto a, d \mapsto abc\}, bd)$ be a D0L system. Then the first few words of $S(H)$ are:

$$bd,\ cdaabc,\ aabccdaa,\ cccdaaabccc,\ \ldots$$

I will also use the shorthands $L(\varphi, s)$ and $S(\varphi, s)$ instead of $L((\varphi, s))$ and $S((\varphi, s))$.

1.4 Chomsky hierarchy

A language is *regular* if it is accepted by a finite automaton. A language is *context-free* if it is accepted by a pushdown automaton. A language is *context-sensitive* if it is accepted by a linear bounded automaton. A language is *recursively enumerable* if it is accepted by a Turing machine. A language (resp. a decision problem) is *recursive* (resp. *decidable*), if it and its complement are recursively enumerable.

The definitions of the corresponding automatons are omitted for brevity.

1.5 Graphs

A *graph* G is a pair $(V,\ E)$, where V is a set, whose elements are called *vertices*, and E is a set of unordered pairs of distinct vertices, whose elements

are called *edges*. The vertices of an edge are its *endpoints*. A *directed* graph is a graph whose edges are ordered. A graph *with self-loops* is a graph, where the endpoints of an edge need not be distinct. Sometimes, edges of a graph have various *weights* and *labels* assigned to them.

The *degree* of a vertex is the number of edges of whose it is an endpoint. In a directed graph, the *outdegree* (resp. *indegree*) of an vertex is the number of edges of whose it is the first (resp. second) endpoint.

A *path* is a sequence of distinct vertices $v_1, v_2, \ldots, v_n$ such that for each integer $1 \leq i \leq n-1$ there exists an edge (v_i, v_{i+1}). A *cycle* is a sequence of vertices $v_1, v_2, \ldots, v_n$ such that $v_1, v_2 \ldots v_{n-1}$ is a path, $v_1 = v_n$ and there exists the edge (v_{n-1}, v_n). A *conjugate* of a cycle is defined analogously to the conjugate of a word.

A *subgraph* of a graph G is a graph whose vertices and edges are subsets of the vertices and edges of G. A *component* of a graph G is a subgraph of G such that for each pair of distinct vertices u and v there is a path from u to v and no additional vertex or edge from G can be added. For directed graphs components are instead called *strongly connected components* and a *weakly connected component* of a directed graph G is a subgraph of G, which would be a component of G if G was not directed.

1.6 Note on pseudocode & time complexity analysis

The algorithms discussed in the second chapter are accompanied by their description in pseudocode. This is done partly for clarity, but also to make time complexity analysis easier. The time complexity analysis itself is conducted in the "usual way" – for example, as is described in the book Introduction to Algorithms by Thomas H. Cormen, Charles Eric Leiserson, Ronald Linn Rivest and Clifford Seth Stein [19].

To elaborate, since I am interested in the difference between, for example, linear and quadratic time and not just between polynomial and nonpolynomial time, the analysis is conducted using a sequential random-access machine, where accessing a memory cell with an arbitrary index takes $O(1)$ time in the worst-case. From now on, whenever a bound is mentioned, it will be implicitly in the worst-case, unless specified otherwise. Usually a lower bound can also be derived, but I will only mention the upper bounds for simplicity's sake.

Furthermore, it is assumed without any perceived loss of generality that the morphism on input has an integer alphabet from 1 to n, and that it is rep-

resented as an array of pointers to arrays, where the i-th pointer points to the array representing the image of the letter i. It is also assumed that the operation of checking the length of an array takes $O(1)$ time, and that input is not to be modified (it is read-only). Axiom, if any, is also an array.

The size of the morphism will be measured using the following: Let n denote the number of letters of the morphism, m the sum of the lengths of all images of the morphism, and l the length of the maximal image. In addition, let empty images have length one when counting m, so that for example $O(n+m)$ can be simplified to $O(m)$.

Finally, memory complexity is generally not mentioned, unless it grows superlinearly (even if the optimal value is constant), since in practice linear memory complexity is usually not problematic. However, superlinear memory complexity is noted, since then the time complexity also becomes superlinear due to having to allocate the memory.

Let me give an example of a simple algorithm, which will also be useful later. It has as an input a D0L system and returns which of its letters are reachable. Let letters of a D0L system mean the letters of its corresponding morphism. A letter of a D0L system is *reachable* if it occurs in any word in the language of the system, otherwise it is *unreachable*. For example, let $H = (\{1 \mapsto 123, 2 \mapsto 34, 3 \mapsto \varepsilon, 4 \mapsto 2, 5 \mapsto 1\}, 232)$ be a D0L system, then $L(H) = \{232, 3434, 22\}$ and the letters 2, 3, and 4 are reachable and the letters 1 and 5 are unreachable. The pseudocode is available in Algorithm 1.1.

Algorithm 1.1 Reach

Input: D0L system G (morphism φ with letters $1 \dots n$ and axiom s)
Output: array R, marking which letters of G are reachable

```
 1: R ← the resulting array, by default all values are set to false
 2: todo ← an empty stack
 3: for i ← 1 to |s| do
 4:     if R[s[i]] = false then
 5:         R[s[i]] ← true
 6:         push s[i] onto todo
 7: while todo is not empty do
 8:     a ← pop from todo
 9:     for i ← 1 to |φ[a]| do
10:         b ← φ[a][i]
11:         if R[b] = false then
12:             R[b] ← true
13:             push b onto todo
14: return R
```

Each letter is pushed onto the stack at most once, and $O(l)$ time is spent on each letter when it is popped. At the start we also spend $O(|s|)$ time on the axiom, thus the total time complexity is $O(|s| + nl)$. However, when we take into account the whole run of the algorithm, it can be seen that this is too pessimistic (each letter looks at its own image). Therefore we obtain a slightly tighter bound $O(|s| + n + m) = O(|s| + m)$, which is linear with regards to the size of the input.

Chapter 2

Repetitiveness of D0L languages

We shall require some factor-oriented notation.

Definition. Let L be a language. A word v is a *factor* of L, if there exists some word w in L such that v is a factor of w. The set of all factors of L is denoted by $F(L)$.

For example, see the D0L system $H = (\{a \mapsto aa, b \mapsto bb, c \mapsto cc\}, abc)$, where $L(H) = \{a^{2^n} b^{2^n} c^{2^n} \mid n \geq 0\}$ and it can be seen that

$$\begin{aligned} F(L(H)) = &\{\varepsilon\} \cup \{a^i \mid i \geq 1\} \cup \{b^j \mid j \geq 1\} \cup \{c^k \mid k \geq 1\} \\ &\cup \{a^i b^j \mid i \geq 1 \wedge j \geq 1\} \cup \{b^j c^k \mid j \geq 1 \wedge k \geq 1\} \\ &\cup \{a^i b^{2^j} c^k \mid i \geq 1 \wedge j \geq 0 \wedge k \geq 1 \wedge i \leq 2^j \wedge 2^j \geq k\}. \end{aligned}$$

The shorthand $FL(G)$ will be used instead of $F(L(G))$ in the rest of this thesis.

The following two definitions are crucial in the context of this chapter. They were put forward by G. Rozenberg and A. Ehrenfeucht in their paper published in 1983 [4].

Definition. Language L is *repetitive* if for each integer $k \geq 1$ there is some nonempty word w such that the word w^k is a factor in L.

Language L is *strongly repetitive* if there is some nonempty word w such that for each integer $k \geq 1$ the word w^k is a factor in L.

It is clear that strong repetitiveness implies repetitiveness. Conversely, let me give an example of a language that is repetitive but not strongly repetitive.

Recall from the previous chapter, that $T = \{0 \mapsto 01, 1 \mapsto 10\}$ is the Thue-Morse morphism and that for all integers $i \geq 0$ the $(i+1)$-th word in $S(T, 0)$ is denoted by t_i. Then, the language $\{(t_i 2)^i \mid i \geq 1\}$ is a repetitive but not a strongly repetitive language. For clarity, here are the three smallest words from this language:

$$012$$
$$0110201102$$
$$01101001201101001201101001 2$$

Informal proof: For all integers $i \geq 0$ the words t_i are prefixes of the Thue-Morse word. The Thue-Morse word is cube-free, hence it and also all its prefixes must also be k-power free for all integers $k > 3$. There also cannot be any cubes created on the boundaries of neighbouring t_is, since they are delimited by 2s, which occur nowhere else. Thus the higher powers are generated only by the explicit power operator in the definition.

Note that this language is also context-sensitive (it can be seen that its definition can be checked with a linearly bounded automaton). However, if we limit our expressive power more, it becomes quite hard to generate such an example. For example, from the well-known pumping lemma for regular languages it can be seen that a regular language is infinite if and only it is strongly repetitive, and since repetitiveness implies infiniteness, it can be seen that a regular language is repetitive if and only if it is strongly repetitive. For analogous the analysis is analogous thanks to the pumping lemma for context-free languages.

However, for D0L languages the situation is different, as there is no equivalent of the pumping lemma for D0L languages. When a D0L language is infinite, it does not have to be repetitive. For example, see $L(T, 0)$. Nonetheless, in the same paper G. Rozenberg and A. Ehrenfeucht managed to prove that the situation is the same after all (and hence also the language above is not D0L):

Theorem 2.1 (G. Rozenberg and A. Ehrenfeucht [4] – Theorem 2)**.** *A D0L language is repetitive if and only if it is strongly repetitive.*

This means that to find a factor in a D0L language, which is an arbitrarily large power, we can focus on finding a word satisfying the definition of strong repetitiveness. G. Rozenberg and A. Ehrenfeucht used similar thinking to also prove the following:

Theorem 2.2 (G. Rozenberg and A. Ehrenfeucht [4] – Theorem 1)**.** *It is decidable whether a language generated by an arbitrary D0L system is repetitive.*

However, the algorithm associated with their proof is quite complicated and has uncertain time complexity. A different algorithm was devised by Y. Kobayashi and F. Otto in their paper published in 2000 [5]. If the size of the alphabet of the morphism of the D0L system is taken to be a constant, then the algorithm runs in polynomial time. A third algorithm, introduced by K. Klouda and Š. Starosta in their paper published in 2015 [6], has the same time complexity considerations, however, instead of returning only a boolean answer, it returns a "description" of all the words satisfying the definition of strong repetitiveness of the studied D0L system.

The rest of this chapter will be spent describing the algorithm by K. Klouda and Š. Starosta. However, before we can move on to that, we also have to describe several other problems and their corresponding algorithms, which the main algorithm makes use of. These include: classifying mortal and bounded letters of a morphism, deciding whether a morphism is injective, and finding a simplification of a morphism.

2.1 Mortal and bounded letters

Let us begin with definitions.

Definition. Let φ be a morphism and a its letter. Then the letter a is *mortal* if ε is in $L(\varphi, a)$, otherwise it is *immortal*. Similarly, the letter a is *bounded* if $L(\varphi, a)$ is finite, otherwise it is *unbounded*.

For example, see the morphism $\psi = \{1 \mapsto \varepsilon, 2 \mapsto 2, 3 \mapsto 33\}$. The letter 1 is mortal and bounded, since $L(\psi, 1) = \{\varepsilon\}$. The letter 2 is immortal but bounded, since $L(\psi, 2) = \{2\}$. Finally, the letter 3 is immortal and unbounded, since $L(\psi, 3) = \{3^{2^k} \mid k \geq 0\}$.

(Im)mortal and (un)bounded words are defined analogously.

The nomenclature for these terms is not exactly set in stone in the surrounding literature. While K. Klouda and Š. Starosta use the names above [6], for example P. M. B. Vitányi calls immortal letters vital [20], A. Ehrenfeucht and G. Rozenberg call immortal letters alive and bounded as having rank zero [4], B. Lando calls bounded letters finite and unbounded infinite [21] and J. Cassaigne and F. Nicolas call unbounded letters growing [14].

There already is an algorithm for finding bounded letters in SageMath. It was implemented by Š. Starosta and it is based upon linear algebra and cyclotomic polynomials. However, the algorithm described will be based upon a different

algorithm by Y. Kobayashi and F. Otto, which they devised in the proof of the following proposition:

Proposition 2.3 (Y. Kobayashi and F. Otto [5] – Proposition 2.6)**.** *It is decidable in polynomial time whether a letter of a morphism is bounded.*

I will show that their algorithm can be modified to run in linear time with regards to the size of the morphism. Contrary to this, the algorithm based upon linear algebra requires turning the morphism into a certain matrix of size $n \times n$, which takes $O\left(n^2 + m\right)$ time, and then manipulating it further. For example, it requires the computation of the characteristic polynomial of the matrix, which has time complexity $O\left(n^3\right)$.

2.1.1 Mortal letters

Since it is the easier task of the two, let us start with classifying mortal letters. The following lemma characterizes morphisms with mortal letters:

Lemma 2.4. *Let φ be a morphism. φ has at least one mortal letter $\iff$ φ is erasing.*

Proof. $\Longleftarrow$: The mortal letter is the letter with the empty image.

$\Longrightarrow$: Let a be the mortal letter of the morphism. Then there must be an integer $n \geq 1$ such that $\varphi^n(a) = \varepsilon$. Let n_0 be the smallest such integer. If $n_0 = 1$, then $\varphi(a) = \varepsilon$ and the implication holds, thus let us assume $n_0 > 1$. Then there exists a nonempty word w such that $\varphi^{n_0-1}(a) = w$ and $\varphi(w) = \varepsilon$, hence the letters from which w is made of must have empty images. □

We can easily find all the letters in a morphism with an empty image. Let us denote the set of these letters as M_1. However, M_1 is not equal to the set of mortal letters. We also have to find letters (let us denote their set M_2), whose image contains only letters from M_1, and not only that, we also have to find letters (M_3), whose image contains only letters from M_1 or M_2, and so forth. Let $\varphi : A^* \to A^*$ be a morphism. Y. Kobayashi and F. Otto formally described these sets like this:

$$M_1 = \{a \in A \mid \varphi(a) = \varepsilon\}$$
$$i \geq 2\colon M_i = M_{i-1} \cup \{a \in A \setminus M_{i-1} \mid \varphi(a) \in M_{i-1}^+\}$$

It can be seen, that if $M_i = M_{i+1}$, then $M_i = \bigcup_{j \geq i} M_j$. Furthermore, since by Lemma 2.4 there is no other way for a letter to become mortal, it follows that the set of mortal letters (let us denote it by M) is equal to $M_{|A|}$. From

this Y. Kobayashi and F. Otto conclude, that the set M can be found in time polynomial with regards to the size of the morphism.

I will expand slightly upon their statement. We can find the set M using the following algorithm: Repeatedly remove letters with empty images and all their occurrences in the images of other letters from the morphism until there is nothing left to remove. The removed letters are mortal, the rest are immortal.

Algorithm 2.1 MortalLetters

Input: morphism φ_{input} with letters $1 \dots n$
Output: array M, marking which letters of φ are mortal

```
 1: φ ← a copy φ_input, so that we can modify it
 2: M ← the resulting array, by default all values are set to false
 3: done ← false
 4: while done = false do
 5:     done ← true
 6:     for a ← 1 to n do
 7:         if M[a] = true then
 8:             continue
 9:         if φ[a] is empty then
10:             M[a] ← true
11:             remove all occurencess of a in images of φ
12:             done ← false
13:             break
14: return M
```

The algorithm is described in pseudocode in Algorithm 2.1. The while loop on line 4 is repeated only when a letter is removed, thus it is executed at most n times. To remove a letter we must first find a letter with an empty image (takes $O(n)$ time), mark it as mortal (line 10, takes $O(1)$ time) and we also have to remove all its occurences from the images of the morphism (line 11, takes $O(m)$ time). In total we get time complexity $O(n \times (n+m))$ $= O(n^2 + nm) = O(nm)$.

We can achieve better time complexity by optimizing the amount of time spent in line 11. We will do this with some preprocessing. First, we change the images in the morphism from arrays to (doubly) linked lists. This allows us to remove an occurrence in $O(1)$ time if we have a pointer to it. Then we create another (singly) linked list for each letter, whose elements point to all the occurrences of these letters in the images of other letters. Let us call the array containing these linked lists *occ*. Figure 2.1 shows an example.

With these linked lists, we can remove all occurrences of a letter in all images

$$\frac{\psi}{1 \to 123 \quad 2 \to 33 \quad 3 \to \varepsilon}$$

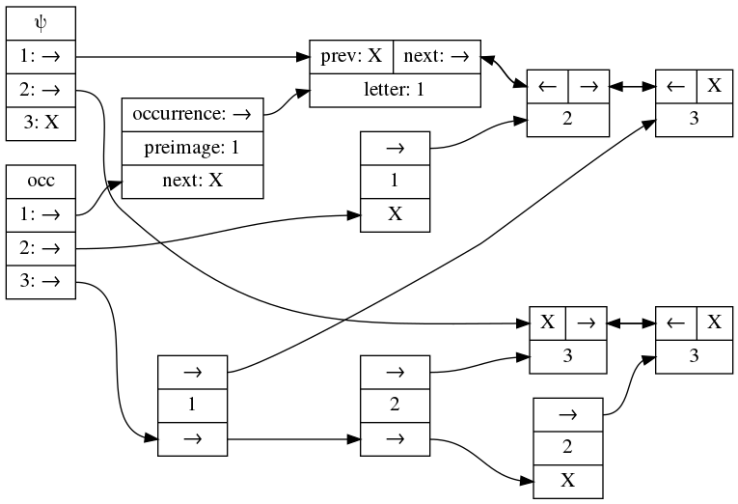

Figure 2.1: Linked lists used in MortalLetters. Descriptions of the fields are omitted on most nodes for clarity.

in time linear with regards to the number of these occurrences. This means, when we take into account the whole run of the algorithm, executing line 11 takes at most $O(m)$ time. This preprocessing can be accomplished in $O(m)$ time. This gives us total time complexity $O(n^2 + m)$.

Another possibility for an asymptotic improvement is hidden in the fact, that we restart the search for an empty image after the removal of each letter. Instead, we could search through all the letters only once, and put the problematic letters (the ones with an empty image) in a stack (or a queue, the order after popping is not important). Then we pop a letter from the stack, start removing its occurrences, and after each removed occurrence we check whether the image we just modified is empty, and if so, we push the preimage of this image into the stack (that is why we save the preimages in the nodes of the linked lists of *occ*). We repeat this until the stack is empty. This merges the work of finding empty letters with the work done in line 11. The initial search takes $O(n)$ time, and each letter is put into the stack (removed from the morphism) at most once. In total, there are $O(n)$ letters popped from the stack. On the whole, we achieve linear time complexity ($O(n+m) = O(m)$).

The ideas from the last two paragraphs are described more concretely in pseudocode in Algorithm 2.2.

Algorithm 2.2 MortalLetters (improved)

Input: morphism φ_{input} with letters $1 \dots n$
Output: array M marking which letters of φ are mortal

```
1: φ and occ ← arrays with the linked lists described in the text
2: M ← the resulting array, by default all values are set to false
3: stack ← an empty stack
4: for a ← 1 to n do
5:     if φ[a] is empty then
6:         push a onto stack
7: while stack is not empty do
8:     a ← pop from stack
9:     M[a] ← true
10:    for each occurrence in occ[a] do
11:        remove the occurrence from the image and
12:            if this was its last letter, push the preimage to stack
13: return M
```

2.1.2 Bounded letters

Now, we can move on to the main part Y. Kobayashi's and F. Otto's proof of Proposition 2.3. It is put together from two auxiliary lemmas. The first of these follows:

Lemma 2.5. *Let φ be a morphism and let $\overline{\varphi}$ be the morphism created from φ by removing all mortal letters of φ (and their occurrences in the images of other letters) and let a be a letter of $\overline{\varphi}$. Then a is a bounded letter of φ if and only if a is a bounded letter of $\overline{\varphi}$.*

For the purposes of the second lemma they define the following directed graph (with self-loops) with weighted edges for a morphism $\overline{\varphi}$ (let us denote it by $G^{all}_{\overline{\varphi}}$ for a morphism $\overline{\varphi}$), where vertices are letters of $\overline{\varphi}$ and there is an edge from a vertex a to a vertex b with a weight k if the letter b occurs in the image of the letter a and the image of the letter a is of length k. For example, see Figure 2.2. Now, the second lemma follows:

Lemma 2.6. *Let $\overline{\varphi}$ be a morphism without mortal letters and a its letter. Then a is unbounded if and only if in the graph $G^{all}_{\overline{\varphi}}$ a cycle containing an edge of weight larger than one can be reached from the vertex a.*

Y. Kobayashi and F. Otto then conclude their proof of Proposition 2.3 by saying that this is checkable in polynomial time. In the rest of this section, I will describe a modified version of their algorithm. For the sake of completeness, its correctness will be proven from the ground up.

ψ

$a \mapsto b$	$b \mapsto c$	$c \mapsto a$
$d \mapsto bc$	$e \mapsto de$	$f \mapsto dg$
$g \mapsto hi$	$h \mapsto g$	$i \mapsto i$

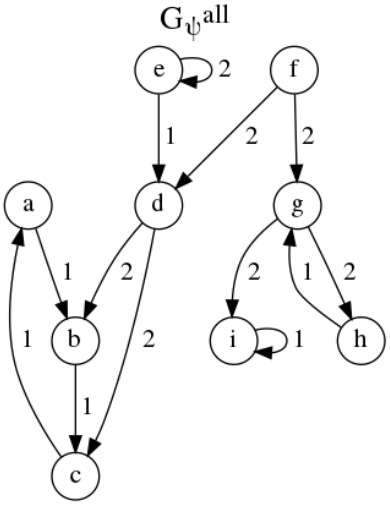

Figure 2.2: G^{all} graph.

Let a single-letter word be a shorthand for a word of length one. The following lemma shares certain similarities with Lemma 2.4:

Lemma 2.7. *Let φ be a morphism. If φ has at least one bounded letter, then at least one letter in φ has an empty or a single-letter image.*

Proof. Let us prove the contraposition: if each letter in φ has an image of length at least two, then φ has no bounded letters. With such a morphism, the image of any word is always strictly longer than the word itself. Thus, for any letter a from φ, each word in $S(\varphi, a)$ is unique, and hence $L(\varphi, a)$ must be infinite. □

Unfortunately, unlike Lemma 2.4, the converse is not true. For example, the morphism $\psi = \{1 \mapsto 11, 2 \mapsto 1\}$ has one single letter image, but no bounded letters ($L(\psi, 1) = \{1^{2^k} \mid k \geq 0\}$, $L(\psi, 2) = L(\psi, 1) \cup \{2\}$) . At least the lemma tells us, that we should focus on empty and single letter images. The following lemma is a good start:

Lemma 2.8. *Let a be a letter of a morphism φ. If a is mortal, then a is bounded.*

$$\psi$$

$1 \mapsto 24$	$2 \mapsto 35$	$3 \mapsto 16$
$4 \mapsto 5$	$5 \mapsto 6$	$6 \mapsto \varepsilon$
$7 \mapsto 8$	$8 \mapsto 27$	$9 \mapsto 41$

$$S(\psi, 9) = 9, 41, 524, 6355, \underline{1666, 24, 355, 1666, \ldots}$$

Figure 2.3: Periodic D0L sequence.

Proof. If a is mortal, then there is an integer $n_0 \geq 0$ such that $\varphi^{n_0}(a) = \varepsilon$. Then for all integers $n \geq n_0$ we have $\varphi^n(a) = \varepsilon$, hence $L(\varphi, a)$ must be finite. □

And the following lemma gives us insight into finding the rest of the bounded letters.

Lemma 2.9. *Let a be a letter of a morphism φ. $L(\varphi, a)$ is finite $\iff$ $S(\varphi, a)$ is eventually periodic.*

Proof. $\Longleftarrow$: $S(\varphi, a)$ being eventually periodic means that there exists an integer $s \geq 0$ and an integer $t \geq 1$ such that for all integers $i \geq s$ we have $(S(\varphi, a))_i = (S(\varphi, a))_{i+t}$, hence $L(\varphi, a) = \{\varphi^n(a) \mid n \leq s+t\}$, which is a finite set.

$\Longrightarrow$: Even if $L(\varphi, a)$ is finite, $S(\varphi, a)$ is still infinite, so a word has to eventually appear twice (for example ε), and due to determinism the sequence will from that point onward start repeating (become periodic). □

For example, see Figure 2.3. To identify bounded letters, we need to identify images that create these repeating sequences of words. This leads us to the following definition:

Definition. A sequence of letters $(a_1, a_2, \ldots, a_k)$ of a morphism φ is a *loop* in φ, if for all integers i in $\{1, \ldots, k\}$ there exist words n_i and m_i in M^* such that $\varphi(a_i) = n_i a_{(i \mod k)+1} m_i$, where M is the set of mortal letters, as defined in the previous subsection.

For example, the letters 1, 2 and 3 form a loop in the morphism in Figure 2.3. The following lemma tells us why these loops are interesting to us:

Lemma 2.10. *Let a be a letter of a morphism φ. If a is part of a loop in φ, then a is bounded.*

Proof. Let $(a_1, a_2, \ldots, a_k)$ be a loop of φ. Let a be an arbitrary letter from this loop. Then there exist words n and m from M^* such that $\varphi^k(a) = nam$. The words n and m are mortal, hence they are bounded (by Lemma 2.8) and $L(\varphi, n)$ and $L(\varphi, m)$ are finite. Clearly, $L(\varphi^k, n)$ and $L(\varphi^k, m)$ must also be finite and it can be seen that $L(\varphi^k, nam)$ is a subset of of the finite language $\{xay \mid x \in L(\varphi^k, n) \wedge y \in L(\varphi^k, m)\}$ and thus it is also finite. Therefore also $L(\varphi^k, a) = \{a\} \cup L(\varphi^k, nam)$ must be finite. Finally, it can be seen that $L(\varphi^k, a)$ being finite means $L(\varphi, a)$ must also be finite, which concludes the proof. □

Finally, we can put together a characterization of morphisms with bounded letters.

Theorem 2.11. *Let φ be a morphism. φ has at least one bounded letter $\iff$ at least one of the following is true:*

(1) φ is erasing,

(2) φ has a loop.

Proof. $\Longleftarrow$: If (1) is satisfied, then there is at least one mortal letter (by Lemma 2.4) and thus at least one bounded letter (by Lemma 2.8). If (2) is satisfied, then there is also at least one bounded letter (by Lemma 2.10).

$\Longrightarrow$: If φ has an empty image, then (1) is satisfied, so let us assume there are no empty images. Then we must "make the boundedness happen" with single-letter images, since by the contraposition of Lemma 2.7, without empty and single-letter images there are no bounded letters. However, our options are rather limited. Let a and b be letters of φ such that $a \neq b$. Single-letter images can be either of the "type" $a \mapsto a$ or of the type $a \mapsto b$. Images of the type $a \mapsto a$ create a loop, thus satisfying (2), so let us assume there are also no images of the type $a \mapsto a$. Now, images of the type $a \mapsto b$ only make a "inherit the boundedness" of b, but on their own they can not create a repetition in $S(\varphi, a)$, *except* for the very specific case, when they are combined with other single letter images into a loop, thus satisfying (2). □

Now, we have almost everything we need to describe the algorithm. Let us denote by T the set of all letters (of some morphism $\varphi\colon A^* \to A^*$) in some

loop and let us call a letter a of φ *loopy* if a is in T. From definitions of M and T, it follows, that M and T are disjoint.

Even though by Theorem 2.11 for φ with bounded letters the set $M \cup T$ is necessarily nonempty, it does not mean that the set of bounded letters is equal to $M \cup T$. Indeed, there is one last set of letters (also disjoint with M and T) interesting to us, let us denote it by V_1. It contains letters whose image contains only mortal and loopy letters and at least one loopy letter (otherwise the preimage would belong to M). Similarly as for mortal letters, we also have V_2, V_3, ...:

$$V_1 = \{a \in A \mid \varphi(a) \in (M \cup T)^* \wedge \exists b \in f(a) \colon b \in T\}$$
$$i \geq 2 \colon V_i = V_{i-1} \cup \{a \in A \setminus V_{i-1} \mid \varphi(a) \in (M \cup T \cup V_{i-1})^* \wedge \exists b \in f(a) \colon b \in V_{i-1}\}$$

That is, $V = V_{|A|}$ are the letters who are not part of a loop themselves, but by iterating on them a loop (or multiple ones) is entered. For example, for the morphism in Figure 2.3 we have $M = \{4, 5, 6\}$, $T = \{1, 2, 3\}$ and $V = \{9\}$ (and letters 7 and 8 are unbounded). The fact that all the letters in the images of V_1 are bounded combined with the following lemma tells us that every letter in V is bounded:

Lemma 2.12. *Let a be a letter from a morphism φ. a is bounded $\iff$ $\varphi(a)$ is bounded.*

Proof. This is due $L(\varphi, a) = \{a\} \cup L(\varphi, \varphi(a))$. □

It follow from Theorem 2.11 and Lemma 2.12 that the set of bounded letters is equal to $M \cup T \cup V$. The algorithm for finding bounded letters is therefore based upon finding these sets. In fact it can be described using the following simple steps:

Algorithm 2.3 BoundedLetters

1: Find and remove mortal letters from the morphism.
2: Do the same for loopy letters.
3: Do the same for mortal letters again.
4: The remaining letters are unbounded.

The third step is the same as the first step, since when we remove mortal and loopy letters from the morphism, the definition of letters in V degenerates to the definition of letters in M. Thus, the algorithm is correct.

We already know from the previous subsection how to find mortal letters, and even how to efficiently remove them using linked lists in time $O(m)$. We can

ψ

$a \mapsto b$	$b \mapsto c$	$c \mapsto ad$	$d \mapsto d$
$e \mapsto f$	$f \mapsto g$	$g \mapsto e$	$h \mapsto hh$
$i \mapsto g$	$j \mapsto i$	$k \mapsto j$	$l \mapsto j$

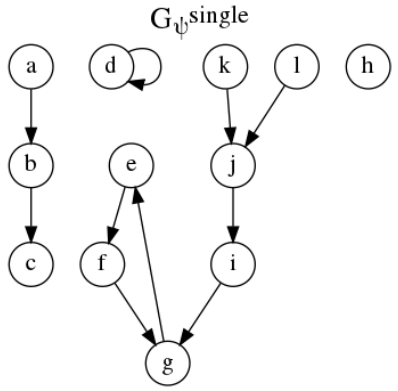

Figure 2.4: G^{single} graph.

efficiently remove loopy letters in the same way, therefore all that needs to be shown now is how to find loopy letters.

First, we can assume the morphism has no mortal letters, since we remove them in the first step, and since loopy letters are immortal, this leaves all of them intact. We start by making a directed graph (with self-loops) (let us denote it G_φ^{single} for some morphism φ), where vertices are letters of φ, and there is an edge from vertex a to vertex b if the letter a has a single-letter image equal to the letter b. Thanks to determinism, each vertex has an outdegree of at most one. Graphs with this property are sometimes called *directed pseudoforests* [22]. For example, see Figure 2.4.

It is easy to see, that cycles in G_φ^{single} correspond to loops in φ.

We can use an algorithm that returns all cycles (without conjugates) in G_φ^{single} to find all loops in φ and thus also all loopy letters. Finding all cycles in an arbitrary directed graph is a nontrivial task, but directed pseudoforests have just a low enough number of cycles and a simple enough structure, that this task can be accomplished easily, as is shown by the following lemma:

Lemma 2.13. *Each weakly connected component (from now on just a component) of a directed pseudoforest has at most one cycle.*

More specifically, if each vertex in a component has an outdegree of 1, then

the component has exactly one cycle, and if at least one vertex in a component has an outdegree of 0, then there can be at most one such vertex and the component has zero cycles.

Proof. The intuitive but informal proof is that if we start walking on some path in the component we can never encounter a fork, we can only encounter another path converging with ours, thus even though there can be multiple starting points in a component, there is exactly one endpoint. This endpoint can be either a vertex with outdegree zero or a cycle. □

The proof of the lemma gives us the algorithm: We repeatedly pick an arbitrary unvisited vertex, and start walking along its path, while also marking visited vertices. We must eventually either find a dead end (which means we are in a component without a cycle) or a vertex we have already visited on some previous path (which means if there is a cycle in this component, we already found it) or a vertex we have already visited on the current path (which means we just found a cycle). We can discern between the second and the third case by keeping a stack of vertices visited on the current path, and when we find a visited vertex, we check if it is in the stack. If it is, we have the third case, otherwise, we have the second case. We can also use the stack for printing the cycle. Since it is impossible for a path to leave a cycle before it gets "stuck" in it, this algorithm is correct and finds all cycles.

The algorithm is described in pseudocode in Algorithm 2.4. Because each vertex is visited only once, and the backtracking to discern between the second and the third case happens only once in each iteration, the algorithm runs in linear time with regards to the size of the graph ($O(n)$). All things considered, this means we have obtained an algorithm for finding bounded letters in linear time ($O(n+m) = O(m)$).

Finally, directed pseudoforests, where each vertex has outdegree of exactly one, are sometimes called functional graphs. The algorithm for finding all cycles in a functional graph is almost exactly the same as Algorithm 2.4, except we can remove lines 10 and 11. The problem of finding all cycles in functional graphs will reappear in later sections.

2.2 Injective morphisms

Injectivity is a well-known concept in mathematics.

Definition. A morphism $\varphi\colon A^* \to B^*$ is *injective* if for all words u and v in A^*, if $\varphi(u) = \varphi(v)$, then $u = v$.

Algorithm 2.4 FindCycles

Input: directed pseudoforest g with vertices $1 \dots n$, represented by an array of size n, which contains in its i-th position the outgoing neighbor of i or 0, if no such neighbor exists

Output: printout of all cycles (without conjugates) in g

```
1:  visited ← array marking which letters were visited, by default all values
    are set to false
2:  for a ← 1 to n do
3:      if visited[a] = true then
4:          continue
5:      history ← an empty stack
6:      while true do
7:          visited[a] ← true
8:          push a onto history
9:          b ← g[a]
10:         if b = 0 then
11:             break                                          ▷ Case 1
12:         if visited[b] = true then
13:             cycle ← an empty stack
14:             while history is not empty do
15:                 c ← pop from history
16:                 push c onto cycle
17:                 if c = b then
18:                     PRINT(cycle)
19:                     break                                  ▷ Case 3
20:             break                                          ▷ Case 2
21:         a ← b
```

However, the negation of this definition will be more useful to us.

Definition. A morphism $\varphi\colon A^* \to B^*$ is *noninjective* if there exist words $u = u_1u_2 \dots u_m$ and $v = v_1v_2 \dots v_n$ in A^* such that $u_1 \neq v_1$, $u_m \neq v_n$ and $\varphi(u) = \varphi(v)$.

For example, the morphism $\psi = \{a \mapsto 11, b \mapsto 12, c \mapsto 123, d \mapsto 31112\}$ is noninjective, since $\psi(bd) = \psi(cab) = 1231112$.

The fact that the definition uses $u_1 \neq v_1$ and $u_m \neq v_n$ instead of $u \neq m$ allows us to not have to repeatedly mention redundant cases in some places in the remainder of this thesis. It does not lose us generality, since the matching letters at the start or the end can be stripped away until we reach unequal letters, which due to the inequality have to be there. It is also worth mention-

ing, that this is one of the few places in this thesis, where we do not require for the alphabet B to be equal to (or a subset of) A.

The problem of morphism injectivity is equivalent to the older problem of unique decodability of a variable-length code, which had plenty of algorithmic attention paid to it. The terminology is slightly different of course: the morphism is called a *code*, its images are called *codewords*, concatenations of codewords are called *messages*, and we are interested whether any message can be *uniquely decoded*. However, I will still use the terminology of combinatorics on words while describing the algorithms.

The only significant difference between morphisms and codes is that it does not make sense for a code to have an empty code word. However, this will not cause any issues to us, as erasing morphisms are trivially noninjective, since for example $\varphi(a) = \varepsilon = \varphi(aa)$. Therefore we can check whether any image of φ is empty at the start of any of these algorithms, and return *false* if so, only taking us $O(n)$ time and $O(1)$ memory, which is guaranteed to be overshadowed by the complexities of the actual algorithm.

To start us off, the eponymous algorithm by August Albert Sardinas and George W. Patterson [23], introduced in their paper published in 1953, is the first and also the most well-known algorithm for deciding the problem of unique decodability of a variable-length code. However, over time better algorithms have emerged, so I will focus on them. Robert Gray Gallager briefly mentioned an improved version of the aforementioned algorithm in an exercise in his textbook on information theory published in 1968 [8]. It was described in much more detail by Shimon Even in his textbook on graph theory published in 1979 [9].

The description of the algorithm I will give here is based upon the informal explanation by R. G. Gallager [8], the pseudocode and proofs by S. Even [9] and the time complexity analysis by M. Rodeh [24]. The algorithm uses the concept of tails (also called dangling suffixes). An example accompanying the explanation of tails is depicted in Figure 2.5.

Let φ be a non-injective morphism and $u = u_1u_2 \dots u_m$ and $v = v_1v_2 \dots v_n$ words satisfying the definition of non-injectivity. Then either $\varphi(u_1)$ must be a prefix of $\varphi(v_1)$ or vice versa. The remaining suffix of $\varphi(v_1)$ (or of $\varphi(u_1)$) is called a tail (t_1 in Figure 2.5). This tail must also either be a prefix of some image of φ, thus creating another tail (t_2) with the suffix of the image, or have some image of φ as a prefix (as is the case for t_2), thus creating another tail with its own suffix (t_3). This holds true, until finally the last tail (t_3) must be equal to some image of φ.

ψ	
$a \mapsto 11$	$b \mapsto 12$
$c \mapsto 123$	$d \mapsto 31112$

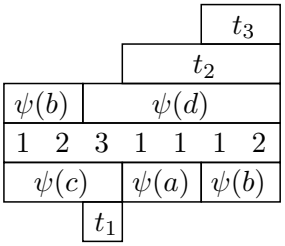

Figure 2.5: Tails for $\psi(bd)$ and $\psi(cab)$.

Let us define the idea of a tail more formally with a definition by S. Even.

Definition. A nonempty word t is a *tail* of a morphism φ if there exist words $c = c_1c_2 \dots c_m$ and $d = d_1d_2 \dots d_n$ such that $\varphi(c)t = \varphi(d)$ and t is a proper suffix of d_n.

The algorithm lies in constructing the set of all tails of a morphism, and checking whether this set contains an image of the morphism. The following lemma by S. Even shows, that this algorithm is correct, and since the proof is short and the concept of tails is relevant in the next section and Chapter 4, I will reword the proof here.

Lemma 2.14 (S. Even [9] – Lemma 4.1)**.** *A morphism is noninjective* $\iff$ *at least one tail (of the morphism) is equal to an image (of the morphism).*

Proof. $\Longleftarrow$: Let a be the letter with an image equal to a tail t and let c and d be the words satisfying the definition of t being a tail. Then the words $u = ca$ and $v = d$ satisfy the definition of noninjectivity of the morphism.

$\Longrightarrow$: Let $u = u_1u_2 \dots u_m$ and $v = v_1v_2 \dots v_n$ be the words satisfying the definition of noninjectivity of the morphism. Then either u_m is a proper suffix of v_n and therefore a tail, or v_n is a proper suffix of u_m and therefore a tail. □

The algorithm is described in pseudocode in Algorithm 2.5. As for correctness, it can be seen that each tail is compared to all images (line 16), so the only thing that remains to be shown is that the algorithm generates all possible tails (if it does not end early). S. Even's proof [9] of this is not as short, but I will still reword it here for the sake of completeness.

Algorithm 2.5 IsInjective

Input: morphism φ with letters $1 \dots n$
Output: boolean saying whether φ is injective

```
 1: if IsErasing(φ) = true then
 2:     return false
 3: todo ← an empty stack
 4: tails ← an empty set
 5: for a ← 1 to n do
 6:     for b ← a + 1 to n do
 7:         if φ[a] = φ[b] then
 8:             return false
 9:         if φ[a] is a prefix of φ[b] OR φ[b] is a prefix of φ[a] then
10:             s ← the corresponding suffix
11:             if s is not in tails then
12:                 add s to tails and todo
13: while todo is not empty do
14:     t ← pop from todo
15:     for a ← 1 to n do
16:         if φ[a] = t then
17:             return false
18:         if φ[a] is a prefix of t OR t is a prefix of φ[a] then
19:             s ← the corresponding suffix
20:             if s is not in tails then
21:                 add s to tails and todo
22: return true
```

Proof. Let φ be a morphism and for every tail t of φ let short(t) be the length of the *shortest* $\varphi(c)$ from all the words c and d satisfying the definition of a tail. The proof uses induction on short(t).

Base case: The smallest possible short(t) happens when $|c| = |d| = 1$, and the corresponding tail is then created on line 9.

Inductive case: Let us assume that all tails p with short$(p) <$ short(t) have been generated already. Let s be a word such that $st = \varphi(d_n)$. s is nonempty since tails are proper suffixes. Then we have the following three possibilities between relationships of s and $\varphi(c_m)$, also depicted in Figure 2.6:

1. $s = \varphi(c_m)$. Then $\varphi(c_m)t = \varphi(d_n)$ and t is created on line 9.
2. s is a proper suffix of $\varphi(c_m)$. Then s is a tail, with short$(s) <$ short(t), hence by the inductive hypothesis it has already been generated and

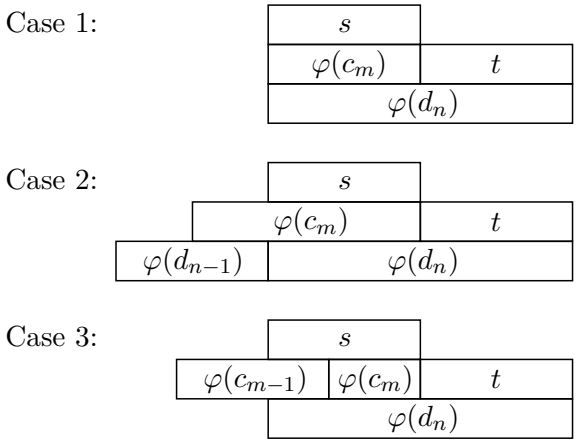

Figure 2.6: Auxiliary diagrams for a proof.

therefore t is created on line 18 with the tail s and the image $\varphi(d_n)$.

3. $\varphi(c_m)$ is a proper suffix of s. Then $\varphi(c_m)t$ is a proper suffix of $\varphi(d_n)$ and thus a tail with $\text{short}(\varphi(c_m)t) < \text{short}(t)$, hence by the inductive hypothesis it has already been generated and therefore t is created on line 18 with the tail $\varphi(c_m)t$ and the image $\varphi(c_m)$.

□

As for time complexity, let us imagine that the set *tails* is implemented using a trie, then searching and/or adding a word of length k takes $O(k)$ time. Then, the while loop on line 5 does $O(n)$ iterations, and it can be seen that each iteration takes $O(n+m) = O(m)$ time, hence the first part of the algorithm (lines from 5 to 12) takes $O(n \times m)$ time in total.

Similarly, each iteration of the for loop on line 13 also takes $O(n+m) = O(m)$ time and its number of iterations is bounded by the number of tails of φ. This number is bounded by m. This is because each tail is also a suffix of an image and since a word of length k has k different suffixes, a morphism has at most m tails. Consequently, the time complexity of the second part of the algorithm (lines from 13 to 21) is $O(m^2)$.

However, what was not considered by M. Rodeh while discussing this algorithm, is the memory complexity. The stack *todo* contains at most all tails, and since each tail is of length at most l, it has a memory complexity of $O(lm)$. However, we can achieve a memory complexity of $O(m)$ if we also add backwards pointers into the trie *tails*, and in the stack we only keep pointers to

the end of the tails in the trie. As we only add tails into the stack whenever we are also adding them to the trie, this should not be a problem.

As for the trie, it also contains at most all tails, and since each tail has size at most l, *and* we need an array of size n for each letter to achieve linear time searching/adding, the trie takes up $O(lnm)$ memory. However, we can also do better. For each tail representing an image there are l tails that are its suffixes. Therefore, if we store the tails in reverse, they will become its prefixes, and each tail representing an image and its l tails will only take up l nodes. Since the tails being stored backwards does not detrimentally affect the algorithm, we get a memory complexity of $O(nm)$ (when we take into account the whole morphism).

Time complexity is no longer affected, and hence the total run time of the algorithm is $O(n \times m + m^2) = O(m^2)$, but this should not stop us, as we can still easily meaningfully improve the memory complexity by compressing the trie. We should only allocate the whole array of size n if a node gains two different successors, otherwise we keep a single pointer. If every image ends on a different letter, there will be only one such array (at the start). If only two images end on a same letter, we will need two such arrays, if three images end on a same letter, we might need three, and so forth. It can be seen, that we will never need more than n such arrays, henceforth we obtain a memory complexity of $O(n^2 + m)$.

There are several other algorithms that achieve better time complexity $O(nm)$. The first one was introduced by Michael Rodeh in a paper published in 1982 [24]. It was however quite complicated, hence two simpler algorithms were described in papers published in 1984, the first of which was written by Alberto Apostolico and Raffaele Giancarlo [25] and the second of which was written by Christoph M. Hoffman [26]. The latter is also notable for making use of different concepts than the tails introduced by A. A. Sardinas and G. W. Patterson. It is unknown whether there is an algorithm with better time complexity than $O(nm)$, but for example linear time complexity seems highly unlikely.

I did not study these algorithms in detail, as I wanted to keep the implementation as simple as possible (to make it easier to integrate it into SageMath) and the algorithm above is fast enough on "practically sized" inputs, especially as the $O(m^2)$ bound is quite pessimistic, since morphisms usually have far fewer tails than m (not to mention that a speedup on practically sized inputs is not guaranteed). It is unknown whether there is an algorithm with better time complexity than $O(nm)$, but for example linear time complexity seems highly unlikely.

2.3 Simplifiable morphisms

A. Ehrenfeucht and G. Rozenberg introduced the notion of a simplification of a morphism in a paper published in 1978 [10], where they used it to solve few problems related to D0L systems. The informal description is as follows: Sometimes it is the case, that the number of letters in a morphism can be reduced, without disrupting the "structure" of the infinite words it generates. The formal definition:

Definition. A morphism $f : A^* \to A^*$ is *simplifiable* if there exist morphisms $h : A^* \to B^*$ and $k : B^* \to A^*$, such that $f = k \circ h$ and $|B| < |A|$, otherwise it is *elementary.* Furthermore, the morphism $g : B^* \to B^* = h \circ k$ is called a *simplification* of f with respect to (h, k).

For example, see the morphism $\psi_f = \{a \mapsto bca, b \mapsto bcaa, c \mapsto bcaaa\}$ (taken from [10]) and let $B = \{x, y\}$. Then ψ_f is simplifiable since there exist for example $\psi_k = \{x \mapsto bc, y \mapsto a\}$ and $\psi_h = \{a \mapsto xy, b \mapsto xyy, c \mapsto xyyy\}$ fitting the definition above. The corresponding simplification is $\psi_g = \{x \mapsto xyyxyyyy, y \mapsto xy\}$. We can see that D0L systems based upon ψ_f and ψ_g generate words with very similar structures, except that the word bc is represented by a single letter x.

2.3.1 Noninjective simplifiable morphisms

Now, how do we find such a simplification? Let us start with the simplest case of an erasing morphism.

Lemma 2.15 (A. Ehrenfeucht & G. Rozenberg [4] – Theorem 1 (i))**.** *If a morphism is erasing, then it is simplifiable.*

I will reword here a proof by Y. Kobayashi and F. Otto [5] (Proposition 3.5) since it says how to find the corresponding morphisms h and k.

Proof. A simplification $g\colon B^* \to B^*$ of a morphism $f\colon A^* \to A^*$, which is erasing, is simply the same morphism with the letters with empty images removed from it (both from preimages and images). Let M_1 denote the set of letters with empty images (as in the previous section). Since $B = A \setminus M_1$, g has a smaller alphabet, hence we only need to find the corresponding morphisms h and k such that $k \circ h = f$ and $h \circ k = f$.

We can take $k\colon B \to A$ such that for all letters b in B we have $k(a) = f(a)$ (that is f with letters from M_1 removed from preimages (but not from images)) and $h\colon A \to B$ such that for all letters b in B we have $h(a) = b$ and for all

letters m_1 in M_1 we have $h(m_1) = \varepsilon$ (identity for letters from B and the empty word for letters from M_1).

Since k has the letters from M_1 removed from preimages, by applying h we also remove them from images, thus obtaining g. Similarly, by applying k to h we get that for all letters b in B we have $k(h(a)) = k(a) = f(a)$ and that for all letters m_1 in M_1 we have $k(h(a)) = k(\varepsilon) = \varepsilon = f(m_1)$, thus obtaining f. □

For example, see the morphism $\psi_f = \{1 \mapsto 123, 2 \mapsto 33, 3 \mapsto \varepsilon\}$. Then we have $\psi_k = \{1 \mapsto 123, 2 \mapsto 33\}$, $\psi_h = \{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto \varepsilon\}$ and finally $\psi_g = \{1 \mapsto 12, 2 \mapsto \varepsilon\}$. As can be seen, the simplification can still be erasing. It seems that it is not possible to find h and k such that the mortal letters are simplified away all at the same time. Moving on, the algorithm is described in pseudocode in Algorithm 2.6. The morphisms h and k are easily created in linear time.

Algorithm 2.6 SimplifyErasing

Input: erasing morphism f with letters $1 \dots n$
Output: morphisms h and k such that $h \circ k$ is a simplification of f

```
1:  h ← an array of words of size n
2:  k ← an empty linked list of words
3:  i ← 1
4:  for a ← 1 to n do
5:      if f[a] = ε then
6:          h[a] ← ε
7:      else
8:          h[a] ← i
9:          i ← i + 1
10:         append a copy of f[a] as a node to k
11: k ← k transformed to an array of words
12: return h and k
```

To describe how to simplify arbitrary morphisms, it is convenient to introduce the definition of a simplifiable multiset of words. This is almost exactly the same notion as that of *elementary languages* introduced by G. Rozenberg and A. Salomaa [15] (page 127). Let the underlying set of a multiset Q be denoted by $\operatorname{supp} Q$.

Definition. A multiset of words X is *simplifiable* if there exists a set of words Z, such that $\operatorname{supp} X \subset Z^*$ and $|Z| < |X|$. Z is called a *simplified set* of X.

That is the number of words in Z is smaller than the one in X, but all

the words in X can still be *reconstructed* by concatenating the words in Z. Having the definition expanded to multisets (instead of only sets) allows us to (sometimes) not have to repeatedly mention the trivial case when the images of two letters of a morphism coincide.

Furthermore, let me use the multiset of images of a morphism as a shorthand for the multiset containing a word w with multiplicity m if the morphism has m letters with the image w. The following lemma tells us that we can talk interchangeably about morphisms and their multisets:

Lemma 2.16. *A morphism is simplifiable $\iff$ its multiset of images is simplifiable.*

Proof. $\implies$: Let $f\colon A^* \to A^*$ be a morphism, X its multiset of images and $g\colon B^* \to B^*$ its simplification with respect to (h, k). If f is erasing, then the set $Z = \operatorname{supp} X \setminus \{\varepsilon\}$ is clearly a simplified set of X.

Let us assume now that f is nonerasing. We will show that a simplified set Z of X is the underlying set of the multiset of images of k. Since k is a morphism from B^* to A^*, we have that $|Z| \leq |B| < |A| = |X|$. Now, we have to show that $\operatorname{supp} X \subset Z^*$. Since f is nonerasing, h must also be nonerasing, otherwise we have $f(a) \neq k(h(a))) = k(\varepsilon) = \varepsilon$, where a is a letter with an empty image under h, which is a contradiction with $f = k \circ h$.

Now, let a be an arbitrary letter of f (or h), w its image under f and v its image under h. We have that $w = k(v)$, hence an arbitrary word from X is equal to the concatenation of several words from Z and thus $\operatorname{supp} X \subset Z^*$. Informally, images of h give us the "description" of how images of f are "reconstructed" from images of k.

$\impliedby$: I will reword here some work of Y. Kobayashi and F. Otto [5] (under Proposition 4.3). Let f be a morphism $A^* \to A^*$, X its multiset of images and Z a simplified set of X. Then let B be an arbitrary alphabet such that $|B| = |Z|$ and k a morphism defined by an arbitrary injective mapping from B to Z. We have that $|B| = |Z| < |X| = |A|$.

Since $\operatorname{supp} X \subset Z^*$, for an arbitrary letter a of f and its image w there must be at least one word v such that $w = k(v)$. If we define h using $a \mapsto v$, then $f = k \circ h$ must hold and we are done. Let us in the future refer to creating h in this way with the shorthand $h = k^{-1} \circ f$, even though inversion of morphisms is not formally defined in this thesis and k might not be injective (and hence k^{-1} not a valid function). □

For example, see the morphism $\psi_f = \{a \mapsto aca, b \mapsto badc, c \mapsto acab, d \mapsto adc\}$ (taken from [6]). Then $\psi_X = \{aca, badc, acab, adc\}$, $\psi_Z = \{aca, adc, b\}$, $\psi_k = \{x \mapsto aca, y \mapsto adc, z \mapsto b\}$, $\psi_h = \{a \mapsto x, b \mapsto zy, c \mapsto xz, d \mapsto y\}$ and finally $\psi_g = \{x \mapsto xxzx, y \mapsto xyxz, z \mapsto zy\}$.

Y. Kobayashi and F. Otto also stated that if f is noninjective, then its multiset of images must be simplifiable (hence f is simplifiable). They did this by invoking a theorem from abstract algebra, but they did not show how to find such a simplifiable set, and the proof of the theorem in [1] (Theorem 1.2.5) does not necessarily show it either. A. Ehrenfeucht and G. Rozenberg also proved that noninjective morphisms are simplifiable [10] (Theorem 1 (ii)) using a distinct proof, but their proof is also nonconstructive.

However, Tero Harju and Juhani Karhumäki gave a procedure for finding it (but without time complexity analysis) in their paper published in 1986 [11] (Section 4). Nonetheless, they also employ a lot of terminology from abstract algebra, and so to avoid having to introduce a nontrivial number of definitions, I will rather give a distinct proof, whilst also explaining the algorithm. First, we need the following definition and a subsequent lemma.

Definition. A set of words is *prefix-free* if no word in it is a prefix of another word in it.

Lemma 2.17. *If a multiset of words is simplifiable, then at least one of its simplified sets is prefix-free.*

Proof. Let X be a multiset of words and Z its arbitrary simplified set. If Z is prefix-free we are done, so let us assume it is not. Then we will employ the following algorithm to create a prefix-free set $\overline{Z}$ such that $|\overline{Z}| \leq |Z|$ and $Z \subset \overline{Z}^*$, hence $\overline{Z}$ is also a simplified set of X.

Let $Z_0 = Z$. Now, we iteratively obtain the set Z_{i+1} from the set Z_i like this: If there are two different words u and w in Z_i such that $w = uv$ for some word v (that is one word in Z_i is a prefix of another word in Z_i), then we arbitrarily pick one such a pair of words u and w and do $Z_{i+1} = Z_i \setminus \{w\} \cup \{v\}$. There must be some integer $j \geq 0$ such that there is no longer such pair of words u and w. Then the set $\overline{Z} = Z_j$ is clearly prefix-free. Now, let us prove $|\overline{Z}| \leq |Z|$ and $Z \subset \overline{Z}^*$ by induction.

Base case: $Z_0 = Z$. Clearly, $|Z| = |Z|$ and $Z \subset Z^*$.

Inductive case: Let us assume $|Z_i| \leq |Z|$ and $Z \subset (Z_i)^*$. Then to create Z_{i+1} we remove the word w and add the word v, as described above. Since w was definitely present in Z_i, we get $|Z_{i+1}| \leq |Z|$ and since the word w can

be "reconstructed" from the concatenation of words u and v, we get $Z_i \subset (Z_{i+1})^*$. □

With this algorithm in hand, we can move on to the next theorem.

Theorem 2.18 (A. Ehrenfeucht and G. Rozenberg [10] – Theorem 1 (ii))**.** *If a morphism is noninjective, then it is simplifiable.*

Proof. If the morphism is erasing, then by Lemma 2.15 it is simplifiable, so let us assume it is nonerasing.

Let f be a noninjective nonerasing morphism, X its multiset of images and Z the result we get by running the algorithm from the proof of the previous lemma on $\operatorname{supp} X$. Then clearly $|Z| \leq |X|$ and $\operatorname{supp} X \subset Z^*$. We just need to show that $|Z| < |X|$ if f was noninjective. Let B be an arbitrary alphabet such that $|B| = |Z|$, k a morphism defined by an arbitrary injective mapping from B to Z and $h = k^{-1} \circ f$.

First, since no image of k is a prefix of another image of k, k has no tails (as described in the algorithm for checking the injectivity of a morphism from the previous section), hence no tail of k is equal to an image of k and therefore by Lemma 2.14 k is injective.

Secondly, we show that if at least two images in h end on the same letter, then $|Z| < |X|$. If two letters a and b of f (or h) have the same image under f, then $|Z| \leq |\operatorname{supp} X| < |X|$ and they must also have the same image under h, since otherwise $h(a) \neq h(b)$ but $k(h(a)) = k(h(b))$ (from $f = k \circ h$), which is a contradiction with k being injective. Hence at least two images in h end on the same letter and the implication holds.

Let us therefore assume that $|\operatorname{supp} X| = |X|$. The rest of the proof of this implication is informal. Let q be a function from A to B defined by $q(a) = b$, where $k(b)$ is a suffix created during the algorithm from the proof of the previous lemma from $f(a)$ by having its various prefixes removed. If for some two letters a_1 and a_2 of f we have $a_1 \neq a_2$ and $q(a_1) = q(a_2)$, then at some point during the algorithm the suffix added to the set must have already been in the set, thus the size of the set decreased. It can be seen, that since the images of h are "descriptions" of how corresponding images of f are "reconstructed" from images of k, that the function q is equal to the function returning the last letter of $h(a)$, thus the implication holds.

Finally, as a proof by contradiction let f be noninjective and u and v two words satisfying the definition of noninjectivity of f but $|Z| = |X|$. Since u

and v end on a different letter and $|Z| = |X|$, $h(a)$ and $h(b)$ also end on a different letter, hence $h(a) \neq h(b)$. However, since $f(a) = f(b)$ and $f = k \circ h$, we also need for $k(h(a)) = k(h(b))$ to hold, but because we have $h(a) \neq h(b)$, this is a contradiction with k being injective. □

The algorithm is described in pseudocode in Algorithm 2.7. The images of the morphism are implemented as a sequence and not as a multiset, however, this does not meaningfully change the algorithm. The important part, that is the removal of prefixes, is contained on lines 1-25. A stack *todo* is used to keep track of letters, whose images have to be compared with all the other images ("both" ways). We also need a boolean array *new*, for the sole purpose of checking whether a letter is already in *todo*, so that we do not push the same letter multiple times.

One iteration of the while loop on line 6 takes $O(m)$ time (we compare an image both ways with all the other images). *todo* start with a size n and a letter is pushed onto it only if an image gets its prefix removed, and since the number of prefixes over all images is bounded by m, the while loop on line 6 goes through $O(n + m) = O(m)$ iterations, therefore its time complexity is $O(m^2)$.

The lines 26-30 only remove all the empty images to put k in the required format, this is done in $O(n)$ time.

Construction of h is taken care of by the lines 31-41. Since k is not only injective, but also its set of the images is prefix-free, we have that not only has $k^{-1} \circ f$ exactly one solution, but we also do not need to use any sort of backtracking to find it.

The for loop on line 36 has time complexity $O(m)$ since we again compare an image with all the other images (this time only one way). Combined with the loops on lines 32 and 35 the time complexity is $O(nlm)$. However, when the whole run of the algorithm is taken into account, the for loop on line 36 is entered $O(m)$ times, thus the time complexity of the three loops is also $O(m^2)$.

In total, we get time complexity $O(m^2)$. It is quite possible that similar to the algorithm for deciding injectivity, this can be reduced further, but reaching linear time complexity again seems unlikely.

Algorithm 2.7 SimplifyNoninjective

Input: nonerasing noninjective morphism f with letters $1 \dots n$
Output: morphisms h and k such that $h \circ k$ is a simplification of f

▷ Create k.
$k_{wip} \leftarrow$ a copy of f
$todo \leftarrow$ an empty stack
$new \leftarrow$ an array of size n, by default all values are set to *true*
for $a \leftarrow 1$ to n **do**
    push a onto $todo$
while $todo$ is not empty **do**
    $a \leftarrow$ pop from $todo$
    $new[a] \leftarrow false$
    if $k_{wip}[a] = \varepsilon$ **then**
        continue
    for $b \leftarrow 1$ to n **do**
        if $k_{wip}[b] = \varepsilon$ **then**
            continue
        if $k_{wip}[a] = k_{wip}[b]$ **then**
            $k_{wip}[b] \leftarrow \varepsilon$
        else if $k_{wip}[a]s = k_{wip}[b]$ for some word s **then**
            $k_{wip}[b] \leftarrow s$
            if $new[b] = false$ **then**
                push b onto $todo$
                $new[b] \leftarrow true$
        else if $k_{wip}[a] = k_{wip}[b]s$ for some word s **then**
            $k_{wip}[a] \leftarrow s$
            push a onto $todo$
            $new[a] \leftarrow true$
            break
▷ Clean up k.
$k \leftarrow$ an empty linked list
for $a \leftarrow 1$ to n **do**
    if $k_{wip}[a] \neq \varepsilon$ **then**
        append $k_{wip}[a]$ as a node to k
k and $N \leftarrow k$ as an array and its size
▷ Create h.
$h \leftarrow$ an array of size n
for $a \leftarrow 1$ to n **do**
    $image \leftarrow$ a copy of $f[a]$
    $h[a] \leftarrow$ an empty linked list
    while $image \neq \varepsilon$ **do**
        for $b \leftarrow 1$ to N **do**
            if $image = k[b]s$ for some word s **then**
                append b as a node to $h[a]$
                $image \leftarrow s$
                break
    $h[a] \leftarrow$ h[a] as an array
return h and k

2.3.2 Injective simplifiable morphisms

Now, let us move on to simplifying injective morphisms. The contraposition of Theorem 2.18 tells us, that if a morphism is elementary, then it is injective. The converse of this statement is not true, as we have seen with the first example morphism in this section, which was injective and simplifiable, and the simplification can be found even with the algorithm for noninjective morphisms.

For example of an injective simplifiable morphism which is not simplifiable using the algorithm for noninjective morphisms, see the morphism $\psi = \{a \mapsto abcc, b \mapsto abcd, c \mapsto abdc, d \mapsto abdd\}$, where $\psi_Z = \{ab, c, d\}$, but no image is a prefix of each other.

The proof of the following theorem tells us how to simplify even injective morphisms:

Theorem 2.19 (A. Ehrenfeucht and R. Rozenberg [10] – Theorem 2 (ii))**.** *The problem of deciding whether an arbitrary morphism is elementary is decidable.*

Proof. Let f be a morphism and X its multiset of images. By Lemma 2.16 we can focus on X. The size of a simplified set of X is bounded by $|X| - 1$. Clearly, if there exists a simplified set Z of X, then there also exists a simplified set $\overline{Z}$ of X such that each its word is bounded by the length of the longest word in X (and also is nonempty), since otherwise the words do not help us in any way in reconstructing words from X and can be removed. Therefore we can simply iterate through the finite number of combinations of possible sets until we find one fitting the definition of a simplified set of X (and if no combination fits the definition, then the morphism is elementary). □

The algorithm is described in pseudocode in Algorithm 2.8. It can be seen that the number of combinations of possible sets is reduced by only focusing on those whose elements are factors of words in X. Clearly, this does not harm correctness (though the number of combinations is still very large, as will be discussed later). The combinations that are not prefix-free are rejected on lines 7-8. By Lemma 2.17, this also does not harm correctness and allows us to again create h without backtracking, thus simplifying the algorithm.

The pseudocode omits some details when compared to the previous algorithms, such as how to enumerate all the combinations. This is because the algorithm is very slow, so these details are not as relevant. The rest of the pseudocode in this chapter will be the same way, as all the remaining algorithms have nontrivial time complexity analysis. However, before discussing the time complexity

Algorithm 2.8 SimplifyInjective

Input: injective morphism f with letters $1 \dots n$
Output: morphisms h and k such that $h \circ k$ is a simplification of f or report of failure if such a simplification does not exist

```
1: factors ← an empty set of words
2: for a ← 1 to n do
3:     for each nonempty factor factor of f[a] do
4:         add PrimitiveRoot(f(a)) to factors
5: for i ← 1 to n − 1 do
6:     for each combination comb of words of size i from factors do
7:         if any word in comb is a prefix of another word from comb then
8:             continue
9:         k ← an arbitrary injective mapping from 1...i to comb
10:        if h ← k^{-1} ∘ f has a solution then
11:            h ← k^{-1} ∘ f
12:            return h, k                          ▷ f is simplifiable
13: report failure                                  ▷ f is elementary
```

of this algorithm I would like to mention two modifications, which further cut down on the number of required combinations.

The first modification is to only add such factors into the set of factors, which are primitive words. This is because if there exists a simplified set Z with periodic words, then clearly the same set with the periodic words exchanged with their primitive roots will also satisfy the definition of a simplified set and hence we only need to check combinations of primitive words.

It can be seen that deciding whether a word w is primitive or even finding its primitive root can be done for example by checking whether $w = v^m$ for some integer $m \geq 1$, where for v we try the prefixes of w in ascending sizes. This could lead to a quadratic run time, for example with $w = a \dots ab$. There is also a more sophisticated algorithm, which has linear time complexity, however, as it is nontrivial and not pertinent to my thesis I will skip it here and leave only references to Lothaire [1] (Problem 1.3.4 & Problem 1.1.3).

The second modification is to only check combinations of size $n-1$. However, when combined with checking only combinations that are prefix-free and with checking only combinations whose elements are factors of words in X, this could cause us to miss out on a solution. For example, see the morphism $\psi_f = \{a \mapsto aa, b \mapsto ab, c \mapsto ba, d \mapsto bb\}$, where $\psi_X = \{aa, ab, ba, bb\}$ and the largest prefix-free simplified set made up of only factors of words in ψ_X is { a, b}, which is of size $n - 2$.

On the other hand, this morphism is quite anomalous, as the letters c and d do not occur in words of X. This could be formally denoted by saying that ψ_f is a morphism from A^* to C^*, where $A = \{a, b, c, d\}$ and $C = \{a, b\}$. However, for this we need to expand the definition of simplifiable morphisms to morphisms where $A \neq B$. This is fortunately quite easy:

Definition. A morphism $f : A^* \to C^*$ is *simplifiable* if there exist morphisms $h : A^* \to B^*$ and $k : B^* \to C^*$, such that $f = k \circ h$ and $|B| < |A|$, otherwise it is *elementary*.

It can be seen, that if C is not a subset of A, we can no longer create a simplification $g = h \circ k$. However, this does not have to bother us, as our algorithm only returns the morphisms h and k.

The reason for expanding the definition is the fact that if $|C| < |A|$ (as was noted by G. Rozenberg and A. Salomaa [15] (page 17)), then it can be seen that the morphism f is trivially simplifiable, with B being an arbitrary alphabet of size $|C|$, k being an arbitrary injective mapping from B to C and $h = k^{-1} \circ f$. The following lemma also pertains to our situation:

Lemma 2.20. *Let $f\colon A^* \to C^*$ be a morphism and X its multiset of words (and all letters from C occur in words of X). If there is a prefix-free simplified set Z_1 of X, whose elements are factors of words of X, that is of size smaller than $|X| - 1$ and there is no set Z_2 with the same conditions that is of size $|X| - 1$, then $|C| < |A| - 1$.*

Proof. Let us prove it by contradiction by saying that $|C| \geq |A| - 1$. Let $i = |C| - |Z_1|$ and $j = |A| - 1 - |Z_1|$. Since $i \geq j$, we can create Z_2 by adding j different letters from C, which do not occur as first letters of words in Z_1, as single-letter words to Z_1. The set Z_2 is then still prefix-free, simplified, its elements are still factors of words of X and it is of size $|A| - 1 = |X| - 1$. □

The lemma tells us that if we first check whether $|C| < |A|$ and accordingly return the trivial simplification, then we can search only combinations of size $n - 1$ without losing correctness. The two modifications are described more concretely in pseudocode in Algorithm 2.9.

Now, on to time complexity. A word of length k has $O\left(k^2\right)$ factors (k positions for both the starting and ending indices). Hence, we can say that the number of factors of the multiset of images of a morphism, and also the cardinality of the set *factors*, is $O\left(n \times l^2\right)$. It should be noted, that this is on average probably a pessimistic estimate, as it ignores the facts that l and the average size

Algorithm 2.9 SimplifyInjective (improved)

Input: injective morphism f with letters $1 \dots n$
Output: morphisms h and k such that $h \circ k$ is a simplification of f or report of failure if such a simplification does not exist

```
1: C ← an empty set
2: for a ← 1 to n do
3:     for i ← 1 to |f[a]| do
4:         add f[a][i] to C
5: if |C| < n then
6:     k ← an arbitrary injective mapping from 1 ... |C| to C
7:     h ← k^{-1} ∘ f
8:     return h, k                                  ▷ f is trivially simplifiable
9: factors ← an empty set
10: for a ← 1 to n do
11:     for each nonempty factor factor of f[a] do
12:         add PrimitiveRoot(factor) to factors
13: for each combination comb of words of size n − 1 from factors do
14:     if any word in comb is a prefix of another word from comb then
15:         continue
16:     k ← an arbitrary injective mapping from 1 ... (n − 1) to comb
17:     if h ← k^{-1} ∘ f has a solution then
18:         h ← k^{-1} ∘ f
19:         return h, k                              ▷ f is simplifiable
20: report failure                                   ▷ f is elementary
```

of images of the morphism might be disproportionate, that images probably share common factors , and that we only care about primitive factors.

The number of checked combinations is then $O\left(\binom{n \times l^2}{n-1}\right)$. We have seen in the previous algorithm that if the set of images of k is prefix-free, then computing $h = k^{-1} \circ f$ takes $O(m^2)$ time. It can be seen, that enhancing this procedure to also report failure if it does not have a solution should not have an impact on the time complexity, and that the time required to generate one combination should be inconsequential compared to the time spent on checking its validity. Similarly, the time required to create the set *factors* should be inconsequential when compared to the time spent on the rest of the algorithm. Overall, we get time complexity $O\left(\binom{n \times l^2}{n-1} \times m^2\right)$.

This time complexity is not very illustrative. Let us at least show, that this is acceptable only for morphisms with very small values of l. For example, if $l^2 = O(1)$, then we have $O\left(\binom{n}{n-1} \times m^2\right) = O(nm^2)$, but if $l^2 = O(n)$,

then we have $O\left(\binom{n^2}{n-1} \times m^2\right) = O\left(\binom{n^2}{n} \times m^2\right) = O\left((1/(2\pi))(en)^{n-1/2} \times m^2\right)$ $= O\left(n^{n-1/2} \times m^2\right)$. The second asymptotic equality can be shown using Stirling's approximation [27].

2.3.3 Injective simplifications

The last topic I will discuss in this section is that of the so-called injective simplifications. It is important to note that a simplification does not have to be elementary or even injective. For example, see the morphism $\psi_f = \{a \mapsto abc, b \mapsto a, c \mapsto bc\}$, where $\psi_Z = \{a, bc\}$ corresponds to the simplification $\psi_g = \{x \mapsto xy, y \mapsto xy\}$ (with respect to morphisms $\psi_h = \{a \mapsto xy, b \mapsto x, c \mapsto y\}$ and $\psi_k = \{x \mapsto a, y \mapsto bc\}$), which is clearly still simplifiable.

However, we can repeatedly call the algorithm for simplifying noninjective morphisms until we get an injective one (we remove at least one letter in each use of the algorithm, so this approach has to be finite). The following is a formal definition:

Definition. Let $(f_0,\ f_1,\ f_2, \ldots, f_t)$, $(h_1,\ h_2, \ldots, h_t)$ and $(k_1,\ k_2, \ldots, k_t)$ be three sequences of morphisms such that f_t is injective and for all integers i in $\{1, \ldots, t\}$ it holds that f_i is a simplification of f_{i-1} with respect to $(h_i,\ k_i)$. Let $f = f_0$, $g = f_t$, $h = h_1 \circ h_2 \circ \cdots \circ h_t$ and $k = k_t \circ \cdots \circ k_2 \circ k_1$. Then g is an *injective simplification* of f with respect to $(h,\ k,\ t)$.

Before giving the algorithm for obtaining an injective simplification, let me highlight here a small but notable difference between simplifications and injective simplifications, which is that the equalities $f = h \circ k$ and $g = k \circ h$ no longer hold and instead we have the equalities given by this lemma.

Lemma 2.21 (Y. Kobayashi and F. Otto [5] – Lemma 4.4)**.** *Let g be an injective simplification of f with respect to $(h,\ k,\ t)$. Then $f^t = h \circ k$ and $g^t = k \circ h$.*

An algorithm for obtaining an injective simplification of a morphism is described in pseudocode in Algorithm 2.10. It mostly follows the definition, except for the fact that if the morphism on input is already injective, a trivial output is returned, for the sake of user-friendliness. When it comes to time complexity, it seems that it would be simply equal to $O\left(nm^2\right)$, due to the algorithm for simplifying a noninjective morphism (which has time complexity $O\left(m^2\right)$) being called at most n times. However, such an analysis ignores the fact that a simplification of a morphism can be larger than the morphism.

Algorithm 2.10 InjectiveSimplification

Input: morphism f with letters $1 \dots n$

Output: if f is noninjective, returns morphisms g, h and k and an integer t such that g is an injective simplification of f with respect to (h, k, t), if f is injective, returns $(f, I, I, 0)$, where I is the identity morphism on $1 \dots n$

```
1:  g ← f
2:  h ← k ← the identity morphism on 1...n
3:  t ← 0
4:  while IsInjective(g) = false do
5:      if IsErasing(g) = true then
6:          h_new, k_new ← SimplifyErasing(g)
7:      else
8:          h_new, k_new ← SimplifyNoninjective(g)
9:      g ← h_new ∘ k_new
10:     h ← h_new ∘ h
11:     k ← k ∘ k_new
12:     t ← t + 1
13: return g, h, k, t
```

Let φ_m denote the sum of lengths of all images of a morphism φ. It can be seen, that if we have two morphisms x and y, then for their composition $z = y \circ x$ we have that z_m is at most $x_m \times y_m$ (the worst case happens with for example $\psi_x = \{a \mapsto aa\}$ and $\psi_y = \{a \mapsto aaa\}$) and that it can constructed in $O(x_m \times y_m)$ time.

It can also be seen, that all the algorithms for simplifying a morphism return h and k such that $h_m \leq f_m$ and $k_m \leq f_m$. Nonetheless, as g is a composition of h and k, we have that $g_m \leq (f_m)^2$. Since when creating an injective simplification we do at most $n-1$ simplifications, we have that $g_m \leq (f_m)^{2^{n-1}}$ and hence the time complexity of finding an injective simplification is $O\left(m^{2^{n-1}}\right)$. For the corresponding morphisms h and k we have a similar bound of $h_m = k_m \leq f_m \times (f_m)^2 \times (f_m)^4 \times \cdots \times (f_m)^{2^{n-2}} = (f_m)^{2^{n-1}-1}$.

These bounds are possibly too pessimistic, as clearly the morphisms of type $\{a \mapsto aa\}$ are not simplifiable, and intuitively simplifications should not be larger than the objects they are simplifying. It might be possible to prove that for each noninjective morphisms there exists an injective simplification which has a size for example at most polynomial with regards to the size of the morphism. Nonetheless, Lemma 2.21 seems to hint that the corresponding morphisms h and k could still "give us trouble".

2.4 Infinite periodic factors

This is the first section in which we have to consider D0L systems and not just morphisms. For example see the morphism $\psi = \{0 \mapsto 00, 1 \mapsto 1\}$. Then $L(\varphi, 0)$ is repetitive, but $L(\varphi, 1)$ is not. However, these D0L systems are quite anomalous, in that they do not "make use" of all the letters of the morphism. To make describing the algorithm more convenient, K. Klouda and Š. Starosta only consider the so-called reduced D0L systems.

Definition. A D0L system is *reduced* if all the letters of its morphism are factors of its language.

It is clear that for an arbitrary D0L system there exists a reduced D0L system that generates the same language, and that the reduced D0L system can be found using Algorithm 1.1 in time linear with regards to the size of the D0L system, so there is no loss of generality.

K. Klouda and Š. Starosta begin the description of the algorithm by giving the following two definitions:

Definition. Let G be a D0L system. The infinite periodic word v^ω is an *infinite periodic factor* of G, if v is nonempty and for all integers $k \geq 1$ the word v^k is a factor of $L(G)$.

The infinite periodic factors v^ω and u^ω of G are *equivalent* if the primitive root of u is a conjugate of the primitive root of v. The equivalence class containing v^ω is denoted by $[v]^\omega$.

It is clear that the language generated by a D0L system is (strongly) repetitive if and only if the system contains at least one infinite periodic factor. The following also holds:

Corollary 2.22 (K. Klouda and Š. Starosta [6] – Corollary 2)**.** *Let G be a D0L system. Then the number of primitive words v, such that v^ω is an infinite periodic factor of G, is finite.*

Their algorithm returns the set of all such primitive words. It is split into two parts: the first part takes care of all infinite periodic factors v^ω, where v contains only bounded letters, and the second part finds the remaining ones (where v contains at least one unbounded letter). Hence, let me refer to the first ones with the shorthand *bounded infinite periodic factors* and to the second ones with the shorthand *unbounded infinite periodic factors*.

$$H = (\psi, \text{a})$$
$$\psi$$

a $\mapsto$ Cab	b $\mapsto$ 1c1	c $\mapsto$ E2bd5	d $\mapsto$ BbaA	
A $\mapsto$ B	B $\mapsto$ C	C $\mapsto$ D	D $\mapsto$ E	E $\mapsto \varepsilon$
5 $\mapsto$ 6	6 $\mapsto$ 7	7 $\mapsto$ 8	8 $\mapsto$ 9	9 $\mapsto$ 5
1 $\mapsto$ 2	2 $\mapsto$ 1			

$S(H)$ = a, Cab, DCab1c1, EDCab1c12E2bd52,
EDCab1c12E2bd52111c1BbaA61, ...

Figure 2.7: Pushy D0L system.

2.4.1 Bounded infinite periodic factors

The following definition was introduced by G. Rozenberg and A. Ehrenfeucht to isolate anomalous types of D0L systems:

Definition. Let $G = (\varphi, s)$ be a D0L system and B the set of bounded letters of the morphism φ. The D0L system G is *pushy* if the set $FL(G) \cap B^*$ is infinite.

In other words, a D0L system is pushy, if its language contains infinitely many factors containing only bounded letters. They also show the following:

Lemma 2.23 (G. Rozenberg and A. Ehrenfeucht [4] – Lemma 2.1 (1))**.** *It is decidable whether a D0L system is pushy.*

The lemma itself is not very relevant to us, but a notion they define during its proof, which they also use to give a different characterization of pushy D0L systems, is crucial:

Lemma 2.24 (G. Rozenberg and A. Ehrenfeucht [4])**.** *A D0L system with a morphism φ is pushy if and only if it satisfies the edge condition, that is if there is a reachable letter a, an integer $k \geq 1$, any word v and a bounded but immortal word u such that $\varphi^k(a) = uav$ or $\varphi^k(a) = vau$.*

An example is in order. This subsection will be illustrated on the D0L system H in Figure 2.7. It can be seen that it satisfies the edge condition with for example a = a, $k = 4$, v = EDCab1c12E2bd52111c1Bb and u = A61, since ψ^4(a) = EDCab1c12E2bd52111c1BbaA61.

How we can use the edge condition to obtain a bounded infinite periodic factor was described in more detail by K. Klouda and Š. Starosta. Let u still be the

bounded and immortal word from the definition of an edge condition, and let us say for now it was the suffix. It can be seen, that by iterating φ^k on a an arbitrary number of times the resulting word will have the following suffix

$$u\varphi^k(u)\varphi^{2k}(u)\varphi^{3k}(u)\cdots .$$

Thanks to u being bounded, if $k = 1$, it can be seen that the suffix will be eventually periodic (see Lemma 2.9), that is there are integers $s_1 \geq 0$ and $t_1 \geq 1$ such that $\varphi^{s_1}(u) = \varphi^{s_1+t_1}(u)$. Fortunately, K. Klouda and Š. Starosta have proven that the suffix will be eventually periodic if k is any integer ≥ 1 [6] (Lemma 4), that is there are integers $s_k \geq 0$ and $t_k \geq 1$ such that $\varphi^{s_k\times k}(u) = \varphi^{(s_k+t_k)\times k}(u)$.

Knowing this, we get that the eventually periodic word (and therefore also the bounded infinite periodic factor) is as follows:

$$xv^\omega = u\varphi^k(u)\cdots\varphi^{(s_k-1)\times k}(u)\left(\varphi^{s_k\times k}(u)\cdots\varphi^{(s_k+t_k-1)\times k}(u)\right)^\omega .$$

If u is a prefix instead, the analysis will be analoguous, except the eventually periodic word will be reversed:

$${}^\omega vx = {}^\omega\Big(\varphi^{(s_k+t_k-1)\times k}(u)\cdots\varphi^{s_k\times k}(u)\Big)\varphi^{(s_k-1)\times k}(u)\cdots\varphi^k(u)u.$$

When implementing this programmatically, since we know some possible values of s_k and t_k must exist, we can find the smallest possible ones by iterating φ^k on u, saving the intermediary results (along with their order) in some set, and ending the search when we get a word we have already seen before.

Again, an example is in order. Let us use the u we had in the last example with H, that is $u = A61$ and $k = 4$. Iterating with ψ^4 on u we get: A61 $\overset{1}{\mapsto}$ E51 $\overset{2}{\mapsto}$ 91 $\overset{3}{\mapsto}$ 81 $\overset{4}{\mapsto}$ 71 $\overset{5}{\mapsto}$ 61 $\overset{6}{\mapsto}$ 51 $\overset{7}{\mapsto}$ 91. We see that $s_k = 2$ and $t_k = 5$, and that we have found a bounded periodic factor $v^\omega = (9181716151)^\omega$.

Therefore if a D0L system satisfies the edge condition, then a D0L system has at least one bounded infinite periodic factor. Together with Lemma 2.24 and the obvious fact, that if a D0L system has at least one bounded infinite periodic factor, then it is pushy, we obtain the following:

Lemma 2.25. *The following three statements are equivalent:*

(1) A D0L system is pushy.

(2) A D0L system satisfies the edge condition.

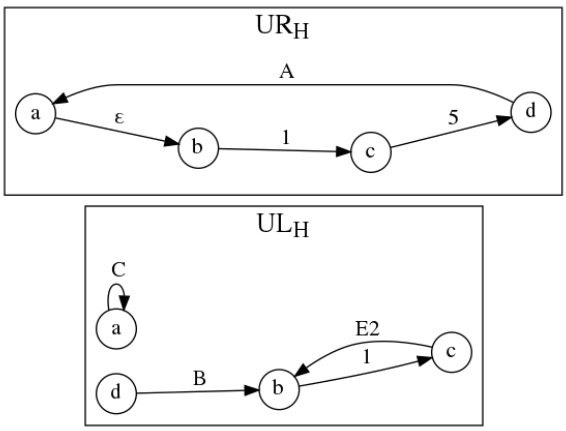

Figure 2.8: Graphs of unbounded letters for the D0L system from Figure 2.7.

(3) A D0L system has a bounded infinite periodic factor.

As a corollary, there is no other way to "generate" a bounded infinite periodic factor, than the one described above.

Now, we need to take a step back, as we still need a way to find all the words u from the definition of the edge condition. For this reason, K. Klouda and Š. Starosta expanded the notion of the edge condition into the so-called graphs of unbounded letters.

Definition. Let G be a (reduced) D0L system with a morphism φ. The *graph of unbounded letters of G to the right*, denoted UR_G, is the labeled directed graph, where vertices are unbounded letters, and there is an edge from the vertex a to the vertex b with the label u, if $\varphi(a) = vbu$, where v is any word and u is a bounded (but not necessarily immortal) word.

The graph of unbounded letters of G to the *left*, denoted UL_G, is defined analoguously, except that the position of u and v in the definition is switched.

For example, see Figure 2.8. From the definition of the graphs of unbounded letters it can be seen that the edge condition is satisfied if and only if one of the graphs of unbounded letters contains a cycle including an edge with an immortal label. Furthemore, let $u_1, u_2, \ldots, u_k$ be the labels of one such a cycle. Then the corresponding word u from the definition of the edge condition is equal to:

$$u_k\varphi(u_{k-1})\cdots\varphi^{k-2}(u_2)\varphi^{k-1}(u_1),$$

if the cycle was from UR. If the cycle was instead from UL, then u is again reversed:

$$\varphi^{k-1}(u_1)\varphi^{k-2}(u_2)\cdots\varphi(u_{k-1})u_k.$$

Coming back to our example, in UR_H we have one cycle of size 4 created from the letters a, b, c, and d. When we start from the letter a, we get $u = \mathrm{A61}$ and find $v^\omega = (9181716151)^\omega$, from b we get $u = \mathrm{B72}$ and find $v^\omega = (5292827262)^\omega$, from c we get $u = \mathrm{1C8}$ and find $v^\omega = (171615191817)^\omega$, from d we get $u = \mathrm{52D}$ and find $v^\omega = (9282726252)^\omega$. In UL_H we have a cycle of size 1 created from only a, and another cycle of size 2 created from b and c. When we start from a we get $u = C$, which is immortal, so we find no bounded infinite periodic factor. When we start from b we get $u = \mathrm{1E1}$ and find $v^\omega = (11)^\omega$, when we start from c we get $u = 22$ and find $v^\omega = (22)^\omega$. We see that cycles that are conjugates of each other can generate bounded periodic factors that are not equivalent.

Since by Lemma 2.25 this is the only way to generate a bounded infinite periodic factor, we must have generated at least one representative v^ω from each class of bounded infinite periodic factors $[v^\omega]$. To return the set of the primitive words v, such that v^ω is a bounded infinite periodic factor, we start by taking an empty set and all the words v found using the algorithm (in the example above 9181716151, 5292827262, 171615191817, 9282726252, 11 and 22), then we make sure to turn all the periodic ones (11 and 22) into primitive ones (1 and 2) by finding their primitive roots, then we add all of them into the set, and finally iterate over each of them and add all of their conjugates to the set ($\{1, 1519181716, 1615191817\ldots\}$).

The vertices of the described graphs are only the unbounded letters since the bounded ones can not satisfy the edge condition (if they could, they would cease being bounded). Thus the image of each vertex contains at least one unbounded letter, and since we can always differentiate the right-most (or left-most) one, the outdegree of each vertex is exactly one. Therefore graphs of unbounded letters are functional graphs, which were described at the end of Subsection 2.1.2, where we have also seen an algorithm for finding all cycles (without conjugates) in a functional graph. We can easily use the output of this algorithm to find all cycles with conjugates.

The algorithm is described in pseudocode in Algorithm 2.11. This is the first algorithm in this thesis in which morphisms are used on multi-letter words, this is denoted in pseudocode by using parentheses instead of brackets.

The time complexity is uncertain, as similar to the algorithm for finding an injective simplification, the algorithm takes significant powers of the morphism (or uses the morphism repeatedly on the same word, which suffers from the

Algorithm 2.11 BoundedInfinitePeriodicFactors

Input: D0L system G (morphism φ with letters $1 \dots n$ and axiom s)
Output: set V, containing all the primitive words v such that v^ω is a bounded infinite periodic factor of G

$\varphi \leftarrow \text{Reduce}(\varphi, s)$
$V \leftarrow$ an empty set of words
$M \leftarrow \text{MortalLetters}(\varphi, s)$
$UR \leftarrow$ the graph of unbounded letters to the right of φ
for *cycle* in $\text{FindCycles}(UR)$ **do**
    if each label in *cycle* is mortal **then**
        continue
    $\varphi^k \leftarrow \varphi^{|cycle|}$
    for each conjugation $\overline{cycle}$ of *cycle* **do**
    ▷ Create u.
        $u \leftarrow$ an empty word
        for for each label *label* in $\overline{cycle}$ **do**
            $u \leftarrow \text{Concatenate}(label, \varphi(u))$
        ▷ Compute s_k and t_k.
        history $\leftarrow$ an empty associative map of pairs (word, integer)
        $u_k \leftarrow u$
        $i \leftarrow 0$
        while u_k is not in *history* **do**
            add the pair (u_k, i) to *history*
            $u_k \leftarrow \varphi^k(u_k)$
            $i \leftarrow i + 1$
        $s_k \leftarrow history[u_k]$
        $t_k \leftarrow i - s_k$
        ▷ Create v.
        $u_k \leftarrow \varphi^{k \times s_k}(u)$
        $v \leftarrow u_k$
        for $\_ \leftarrow 1$ to $(t_k - 1)$ **do**
            $u_k \leftarrow \varphi^k(u_k)$
            $v \leftarrow \text{Concatenate}(v, u_k)$
        for for each conjugate c of v **do**
            add c to V
$UL \leftarrow$ the graph of unbounded letters to the left of φ
do the same steps with UL as with UR above, except the order of the arguments in both calls to the Concatenate function is swapped
return V

same blowup), which have the same problems (and by definition they even are the same) as compositions of morphisms. That is, the time complexity could be approximated by $O(x \times m^{2^y})$, where x is the number of bounded infinite periodic factors and y an unknown function dependant upon the morphism. It might be possible that this time bound is again too pessimistic since we are using the morphism only on bounded words, therefore we can restrict the domain of the morphism to only bounded words, and such morphisms should intuitively not increase in size as much when we take their power.

However, it can be seen that if we only want to decide whether a D0L system is pushy, we only have to find a cycle containing an edge with an immortal label (and therefore we can also ignore conjugate cycles). As reducing a D0L system takes $O(n + m + |s|)$ time (Algorithm 1.1), finding all the unbounded letters takes $O(n + m)$ time (Algorithm 2.3), creating the graphs of unbounded letters can be easily done in $O(n + m)$ time, finding in them all cycles (without conjugates) takes $O(n)$ time (Algorithm 2.4) and finally deciding whether a label of size k is immortal takes $O(k)$ time using the array M from Algorithm 2.2 (which runs in $O(n + m)$ time), we have that deciding whether a D0L system is pushy takes time linear with regards to the size of the D0L system.

2.4.2 Unbounded infinite periodic factors

To easily describe how unbounded infinite periodic factors can be found we need yet another definition.

Definition. Let φ be a morphism, then a word w is a *periodic point* of φ (with a period k), if there exists an integer $k \geq 1$ such that $\varphi^k(w) = w$.

Let $\varphi\colon A^* \to A^*$ be a morphism. Usually, we are interested in infinite periodic points, and more specifically, those that can be "started" from a single letter, that is periodic points w with a period k such that there exists an unbounded letter a of φ so that $\varphi^{k\times\omega}(a) = w$ (let us denote these as *periodic points with origin in an unbounded letter*).

Similar to purely morphic words, to find these types of periodic points we just have to find all unbounded letters a such that $\varphi^k(a) = av$ for any word v (since a is unbounded v has to be immortal) and an integer $1 \leq k \leq |A|$.

This in turn can be found easily using the so-called graphs of first letters – as the name suggests it is a directed graph where vertices are the letters of the morphism φ and there is an edge from vertex a to vertex b if $\varphi(a) = bx$ for any word x.

Since each vertex has an outdegree of at most one, these graphs are directed pseudoforests – finding all cycles in them is easy. After we find a cycle, we also have to check that its vertices are unbounded letters, which is also easy. In fact, it suffices to check only any one of its vertices, since a bounded letter can not have an unbounded letter as its first (or any) letter, thus it can not be in a cycle with one.

How are periodic points related to unbounded infinite periodic factors? If a D0L language has a periodic point w as a factor, *and* w is also an infinite periodic word (that is, $w = v^\omega$, let us denote these as periodic periodic points), then it is not hard to see that it also has an unbounded infinite periodic factor v^ω.

Let us restrict ourselves to D0L systems with an injective morphism for a bit (at the end of the section it will be shown, that this restriction does not matter). K. Klouda and Š. Starosta proved that for D0L systems with an injective morphism the converse of the above also holds:

Theorem 2.26 (K. Klouda and Š. Starosta [6] – Theorem 15)**.** *If a D0L system with an injective morphism has an unbounded infinite periodic factor v^ω, then it also has a periodic periodic point u^ω with an origin in an unbounded letter, where $u \in [v]$.*

That is, in D0L systems with injective morphisms there is a 1-to-1 correspondence between equivalence classes of unbounded infinite periodic factors and periodic periodic points with origin in an unbounded letter. To check that a periodic point w (of an injective morphism $\varphi : A^* \to A^*$) with period p and with an origin in an unbounded letter a is a periodic infinite word, K. Klouda and Š. Starosta employed the following algorithm by Barbara Lando described in her paper published in 1986 [21] (end of Section 3), which can be succinctly described like this: If $\varphi^{p\times k}(a) = azax$, where $k \leq |A|$, x is any word and z is a word containing no occurrences of a (and by [21] (Proposition 3.3) at most one occurrence of each other unbounded letter), then $w = (az)^\omega$ if $\varphi^p(az) = (az)^e$ with $e \geq 2$.

Let me also show a small example: Let $H = (\psi, 1)$ be a D0L system with the injective morphism $\psi = \{0 \mapsto 101, 1 \mapsto 0\}$. Since $\psi^2 = \{0 \mapsto 01010, 1 \mapsto 101\}$, it can be seen that ψ has two periodic points with period 2 with origins in unbounded letters, namely in both 0 and 1, and since H is reduced, these periodic points are also in $FL(H)$. Applying Lando's algorithm to the periodic point with origin in 0 we get that $k = 1, az = 01, ax = 010$ and $\psi^2(01) = 01010101 = (01)^4$, hence $(01)^\omega$ is an unbounded infinite periodic factor of H. Applying Lando's algorithm to the periodic point with origin in 1 we get that

$k = 1, az = 10, ax = 1$ and $\psi^2(10) = 10101010 = (10)^4$, hence $(10)^\omega$ is an unbounded infinite periodic factor of H.

Since by Theorem 2.26 this is the only way to generate an unbounded infinite periodic factor, we must have generated at least one representative v^ω from each class of unbounded infinite periodic factors $[v^\omega]$. Therefore we can proceed in almost the same way as in the previous subsection, except we do not have to find primitive roots, since the word az in Lando's algorithm is already primitive.

Finally, the fact that the above theorem holds only for D0L systems with injective morphisms does not have to bother us thanks to the following lemma by K. Klouda and Š. Starosta based on the work with injective simplifications by Y. Kobayashi and F. Otto [5]:

Lemma 2.27 (K. Klouda and Š. Starosta [6] – Corollary 5)**.** *Let G be a D0L system and let G' be its injective simplification with respect to (h, k, i). Then v^ω is an infinite periodic factor of $L(G)$ if and only if u^ω is an infinite periodic factor of $L(G')$ such that $h(u)^\omega \in [v]^\omega$.*

Thus to find for an arbitrary D0L system G the set of primitive words v, such that v^ω are unbounded infinite periodic factors in G, we can find all such words in the corresponding injective simplification and use k on them. However, since it is possible for k to create periodic words from primitive ones, we need to find their primitive roots and moreover, we also have to again add them into another set and find all their conjugates. Due to this, we can save ourselves some work if we only look for the conjugates once we have used k.

The pseudocode describing this whole subsection is available in Algorithm 2.12. The time complexity is again uncertain since the morphism is iterated a non-constant number of times. The positive difference is that in this case we know that it cannot be iterated more than n times, the negative difference is that in this case we also have to consider unbounded letters.

In the previous subsection, we have seen that a D0L system is pushy if its language contains at least one bounded infinite periodic factor. As a counterpart to that, K. Klouda and Š. Starosta introduced the following definition [13]:

Definition. A D0L system is *unboundedly repetitive* if its language contains at least one unbounded infinite periodic factor.

Now, we can say that a D0L system is (strongly) repetitive if and only if it is pushy or unboundedly repetitive. Based on the pseudocode in this and the

Algorithm 2.12 UnboundedInfinitePeriodicFactors

Input: D0L system G (morphism φ with letters $1 \dots n$ and axiom s)
Output: set V, containing all the primitive words v such that v^ω is an unbounded infinite periodic factor of G

$\varphi \leftarrow \text{Reduce}(\varphi, s)$
$\varphi, \_, k, \_ \leftarrow \text{InjectiveSimplification}(\varphi)$
$V \leftarrow$ an empty set of words
$U \leftarrow \text{UnboundedLetters}(\varphi)$
$G \leftarrow$ functional graph of first letters of φ
for *cycle* in $\text{FindCycles}(G)$ **do**
    if U[the first letter of *cycle*] = *false* **then**
        continue
    $\varphi^q \leftarrow \varphi^{|cycle|}$
    for each letter a of *cycle* **do**
    ▷ Is the periodic point with origin in a a periodic infinite word?
        occurred $\leftarrow$ an array of size n, by default all values are set to *false*
        occurred$[a[i]] \leftarrow$ *true*
        end $\leftarrow 1$
        for $\_ \leftarrow 1$ to n **do**
            prev $\leftarrow |a|$
            $a \leftarrow \varphi^q(a)$
            for $i \leftarrow (prev + 1)$ to $|a|$ **do**
                if $U[a[i]] =$ *true* **then**
                    if *occurred*$[a[i]] =$ *true* **then**
                        end $\leftarrow i$
                        break
                    occurred$[a[i]] \leftarrow$ *true*
            if *end* $\neq 1$ **then**
                break
        if $a[end] \neq a[0]$ **then**
            break
        $v \leftarrow$ substring of a from 1 to $(end - 1)$
        $vq \leftarrow \varphi^q(v)$
        same $\leftarrow$ *true*
        for $i \leftarrow 1$ to $|vq|$ **do**
            if $vq[i] \neq v[i \bmod |v|]$ **then**
                same $\leftarrow$ *false*
                break
        if *same* = *false* **then**
            break
        $v \leftarrow k(v)$
        $v \leftarrow \text{PrimitiveRoot}(v)$
        for for each conjugate c of v **do**
            add c to V
return V

previous subsection, it can be seen that while checking for pushiness (linear time complexity) is easier than finding at least one bounded infinite periodic factor (uncertain time complexity), checking for unbounded repetitiveness is complete only when we finalize the finding of at least one unbounded infinite periodic factor (uncertain time complexity).

One last note regarding the pseudocode: K. Klouda and Š. Starosta proved that if we have an injective morphism φ and an infinite word w such that $\varphi(w)$ is periodic, then w is eventually periodic [6] (Lemma 4). Hence if one letter in a cycle of the graph of first letters is not an origin of a periodic periodic point, then neither are all the other letters in the same cycle and therefore we can use a break statement instead of a continue statement on lines 26 and 35.

Chapter 3

Implementation

The implementation of the algorithms from the previous chapter is split into two patches for SageMath's codebase. The corresponding tickets on SageMath's ticketing system can be found here [28] and here [29] [3]. The Python code can be then viewed by clicking on one of the links in the "Branch" field.

Trying this code out before it is merged in SageMath and before the next minor version of SageMath is released requires one to compile their own version of SageMath. However, Python allows one to easily "monkey patch" the code into the current stable version of Sage during runtime. Therefore I have also included with this thesis a Python file `impl.py` containing a slightly modified version of the implementation, which can be loaded into SageMath using the command `load("/path/to/the/file/impl.py")` [4]. How to use the methods themselves should be clear from the examples in the documentation (`doc.pdf`) for users with at least passing knowledge of SageMath.

The rest of this chapter is structured as follows: first, SageMath is briefly introduced, then some concepts behind its combinatorics on words module are described in more detail, after which a few key differences between the pseudocode and the implementation are highlighted, and finally, the testing methodology is recounted.

[3] It is possible that in the future the larger of these will be split into even more smaller ones to ease up the reviewing process

[4] There is also a text file `sagecell.txt`, which contains a link to a website allowing one to try the methods without even installing SageMath.

Figure 3.1: SageMath's logo. The graph-like symbol has no special meaning [33].

3.1 SageMath

SageMath, previously known as Sage, is a free and open-source mathematics software system [7]. It is licensed under the GPL v3, and runs on Windows, Linux, and OSX. It can also be accessed online on the SageMathCell [30] and CoCalc [31] websites. Some of its more well-known alternatives include Wolfram Mathematica and MATLAB.

SageMath provides a unified interface to many mathematically-minded open-source projects, such as SciPy, Sympy, Maxima, GAP, FLINT, R, and others. Since the release of SageMath's first version in 2005, almost 300 packages have been integrated and over 2.2 million lines of code have been written from scratch [32].

SageMath is primarily written in Python (3), but Cython is also used often, mainly in performance hot spots and for interfacing with the aforementioned libraries. Python is also used as the interface for SageMath itself, though there is a simple preprocessing step involved, which I will describe here for posterity:

- All floating-point constants are wrapped in the `RealNumber` object – to work around the floating-point precision problems.
- All integer constants are wrapped in the `Integer` object – to support faster infinite precision integers than standard Python and to have division between integers return `Rational` objects instead of floats.
- The symbol ^ (xor operator in Python) is replaced (except for in string constants) with ** (power operator in Python), as ^ is conventionally used for the process of taking power of something when writing math on a keyboard, and xor is not as useful in mathematics (it can still be accessed with ^^ since that is replaced with ^).

Now, I will briefly summarise SageMath's development process. SageMath uses git for version control and Trac for tracking issues/tickets and for hosting the git repository (there is also a mirror on GitHub). Informally speaking, one needs to follow these three easy steps to contribute to SageMath (similar as for other open-source projects):

0. Create an account on SageMath's Trac server (the easiest way is by logging through an existing GitHub account).
1. Either create a new Trac ticket or choose an existing one.
2. Push a git branch containing the changed code to the git repository and attach it to the ticket, also set the ticket's status to `needs-review`.
3. The changes will be reviewed (after some indeterminate amount of time) and some objections will probably be raised, which will have to be resolved, but if all goes well, after few such iterations the new code should be merged into the next minor version of SageMath.

3.2 `sage.combinatorics.words`

The part of SageMath that concerns itself with combinatorics on words is called (in Python package naming convention) `sage.combinatorics.words`. The newest version of the source code can be viewed online, for example, at the GitHub mirror [34] of the git repository, and the documentation is available on the SageMath website [35].

All of SageMath makes heavy use of object-oriented design. There are two issues that might crop up when one tries to model words as objects. The first of these is that infinite words require quite different treatment than finite words. That is why SageMath differences between:

- Finite words – these should only use methods that work on finite words.
- Infinite words – these should only use methods that work on infinite words.
- Words of unknown length – these should only use methods that work on both finite and infinite words.

In actuality there are no methods (at least for now) that work on infinite words but do not work on finite ones, so the list can be reworded as such:

- Finite words – these can use all methods.
- Infinite words and words of unknown length – these can only use methods that do not require finite length.

The second issue, orthogonal (theoretically) to the first one, concerns itself with how the word is created, or more precisely, how the word stores and manipulates its own data. SageMath gives us the following options regarding the word's data type:

- Python string.
- Python list.
- Python tuple.
- C array (the fastest way, but also the most limiting).
- Callable (a function or a functor).
- Callable, results of which are cached.
- Python iterator.
- Python iterator, results of which are cached.

Each item in the previous two lists is represented by a class in SageMath. Classes in the first list implement the generic high-level functionality of various algorithms, while classes in the second list implement the specific low-level functionality of storing and viewing data. Finally, each word inherits exactly one class from the first list and one class from the second list. Not every combination is possible though, as is documented in Table 3.1. To summarize what is in the referenced table: anything goes for finite words, infinite words cannot be implemented with finite storage (list, tuple, string, array) and words with unknown length can only be defined with an iterator since callables without a specified length are presumed to be infinite.

There are also classes representing these following languages over some alphabet (the alphabet is inputted by the user):

- Language of all finite words over some alphabet.
- Language of all infinite words over some alphabet.
- Language of all finite and infinite words over some alphabet.

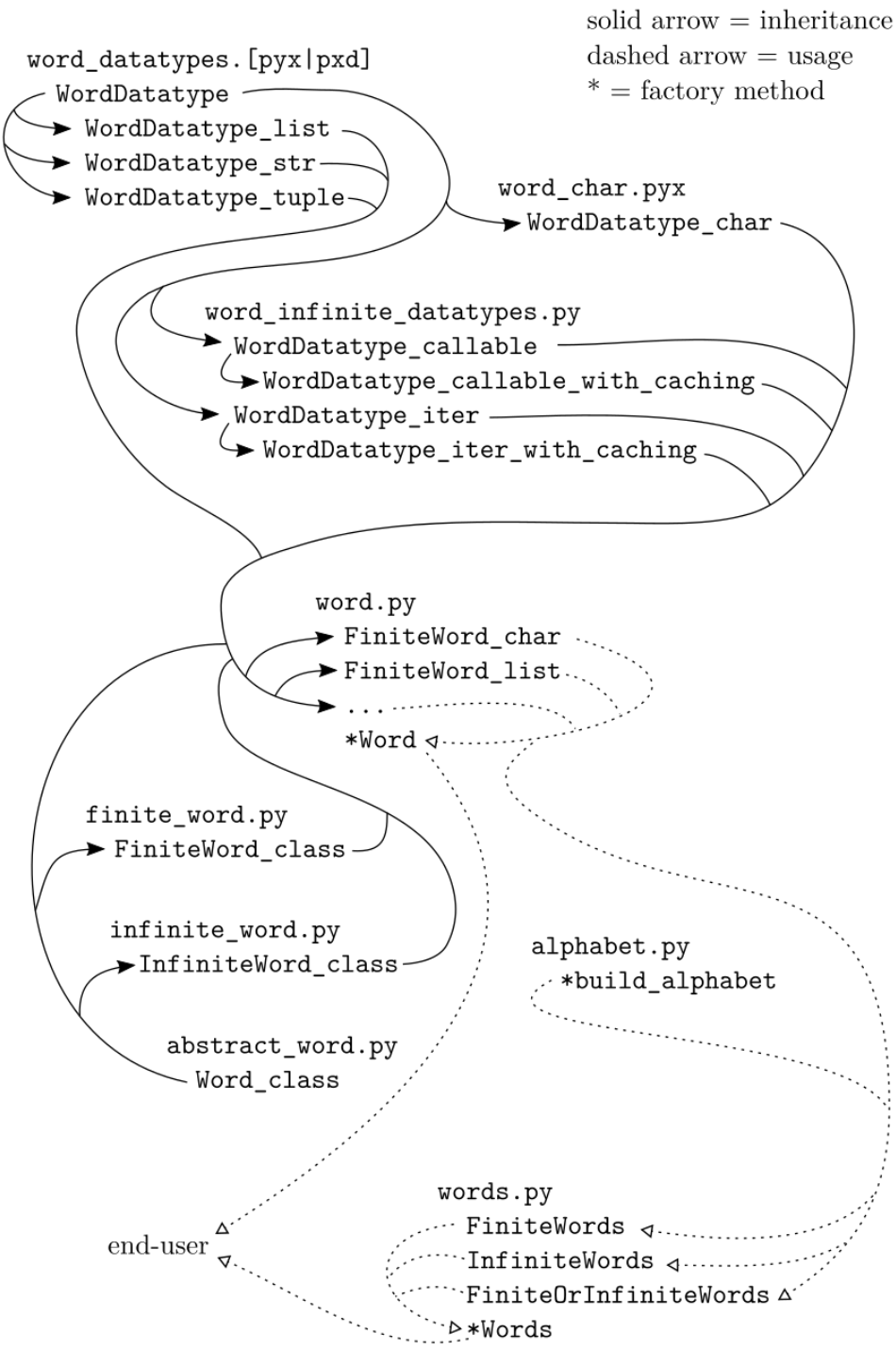

Figure 3.2: Simplified file and class diagram of `sage.combinatorics.words` as of version 9.2.

length → / data type ↓	Finite	Infinite	Unknown
string	✓		
list	✓		
tuple	✓		
array	✓		
callable	✓	✓	
callable (w/ cache)	✓	✓	
iterator	✓	✓	✓
iterator (w/ cache)	✓	✓	✓

Table 3.1: Overview of possible combinations of word classes.

Each word belongs (has a pointer) to one of these languages, depending upon its length. Where finite and infinite words belong is clear, the third language is reserved for words of unknown length. Slightly interesting is that a finite (resp. infinite) word never (directly) belongs to the language of finite and infinite words. It can be sampled from it, but the act of sampling makes it belong directly to the language of finite (resp. infinite) words. This concept is known as *facade* in SageMath development.

These languages are a foundation for a framework that would allow one to easily add more specific languages. However, it is only a foundation, as there is currently no abstract base class `Language` which could be inherited to reduce boilerplate when implementing new languages [5]. There is work being done on a large patch that would add such a class (and which would also refactor most of the code in `sage.combinatorics.words`), however, as of today the patch is not yet completed or more importantly merged into SageMath.

All in all, it is a standard practice in SageMath to have element-like classes belong to parent-like classes and to keep a similar interface across all of SageMath by making them inherit from the generic `Element` and `Parent` SageMath classes. It should be noted that `sage.combinatorics.words` is currently not entirely compliant in this regard (languages inherit from `Parent`, but words do not inherit from `Element`), fixing this is a part of the rework mentioned in

[5] This is not entirely true, as there there is one class called `AbstractLanguage`, however, it is depreciated and also not very useful.

the previous paragraph.

How are the actual classes called and in what files they reside in can be seen in the accompanying diagram in Figure 3.2. These are the files from which the basic skeleton of `sage.combinatorics.words` is made of. There are a few more files providing additional functionality built on top of this skeleton (most important of these being `word_generators.py` and `morphism.py`). Lastly, there are three rather small and specific files: `__init__.py` tells Python this folder is a package, `all.py` specifies what should be imported by default in SageMath and `word_options.py` manages how should the words be printed into the console/file/notebook.

As can be seen in the previous diagram, one is expected to create words either using the `Word` function, or by creating a language over some alphabet using the `Words` function, and then sampling or even enumerating all (or only of certain length) words from such a language. The `words` function (not to be confused with the `Words` function) provides a shortcut for creating some of the more well-known infinite words, such as the Thue-Morse word or the Fibonacci word. Word morphisms are created with the class `WordMorphism`. The morphisms can be applied to both finite and infinite words and also have a few methods of their own, such as computing their fixed points or composition.

Finally, I will end this section with a (nonexhaustive) list of what one can do with finite words in SageMath:

- Compute a power / period / exponent / border / primitive root.
- Find all (or only of certain length) prefixes / suffixes / conjugates / subwords / factors / special factors.
- Match patterns in the word with the Boyer-Moore algorithm.
- Make a suffix tree/trie out of the word and then traverse it as needed.
- Use a surprising number of functions related to palindromes.
- Do a Lempel-Ziv decomposition.
- Check if it is a Lyndon word.
- Do a Burrows-Wheeler transformation.

algorithm	method
Reach (Alg. 1.1)	`reach`
MortalLetters (Alg. 2.2)	`immortal_letters`
BoundedLetters (Alg. 2.3)	`growing_letters`
FindCycles (Alg. 2.4)	–
IsInjective (Alg. 2.5)	`is_injective`
SimplifyErasing (Alg. 2.6)	`simplify`
SimplifyNoninjective (Alg. 2.7)	`simplify`
SimplifyInjective (Alg. 2.9)	`simplify`
InjectiveSimplification (Alg. 2.10)	`simplify_injective`
BoundedInfinitePeriodicFactors (Alg. 2.11)	`infinite_repetitions_bounded`
UnboundedInfinitePeriodicFactors (Alg. 2.12)	`infinite_repetitions_growing`
–	`infinite_repetitions`
–	`is_pushy`
–	`is_unboundedly_repetitive`
–	`is_repetitive`

Table 3.2: Map between algorithms and methods.

3.3 Differences between pseudocode and implementation

Let us start with the high-level differences. Table 3.2 shows the relationships between the algorithms from the previous chapter and the names of the functions they ended up in. As SageMath does not have a class representing D0L systems, all the functions are methods of the `WordMorphism` class, with the methods dealing with D0L systems having the axiom inputted as a parameter.

As can be seen, the name *unbounded* was replaced with the name *growing*, as that was the already the convention in SageMath and the slightly shorter name *infinite repetition* was used instead of *infinite periodic factor*. There is also a slight discrepancy between the definitions in the documentation and this thesis.

Moving on, SageMath already had a method for finding periodic points of a morphism with an origin in an unbounded letter (`WordMorphism.periodic_points`) using the same algorithm I described, which lead to a slightly simpler UnboundedInfinitePeriodicFactors. Similarly, FindCycles was also already implemented in SageMath (`get_cycles` in the `morphism.py` file) using the same algorithm I described, which saved me some work while implementing few of the algorithms.

However, as a result of it not accepting pseudoforests but only functional graphs, the implementation of BoundedLetters is slightly "messier". Speaking of which, BoundedLetters was *also* already implemented in SageMath (`WordMorphism.growing_letters`), as was mentioned in the previous chapter. Nonetheless, as the algorithm described in this thesis is notably faster, I still submitted its implementation (in a separate ticket).

The algorithms SimplifyErasing, SimplifyNoninjective and SimplifyInjective were for the sake of simplicity merged into one method `simplify`. Furthermore, in the pseudocode, the alphabet B of the simplification is always simply a subset of A, but in the implementation, I also added an argument that allows for B to be chosen on input. The alphabets A, C, and B are also called X, Y, and Z instead.

I also added four small auxiliary methods. `infinite_repetitions` simply merges the results of `infinite_repetitions_bounded` and `infinite_repetitions_growing`, while `is_pushy`/`is_unboundedly_repetitive` simply checks whether `infinite_repetitions_bounded`/`infinite_repetitions_growing` has a nonempty output, and finally `is_repetitive` is simply a logical disjunction of `is_pushy` and `is_unboundedly_repetitive`.

Now, the low-level differences. Some constructs were replaced to be more pythonic, for example by using iterators instead of loop counters. The most notable deviation is hidden in the assumption that the morphisms on input have integer alphabets. However, a letter of a word in `sage.combinatorics.words` can be any Python object (and `WordMorphism` itself is implemented using the Python built-in container `dict`, which represents the abstract data type of an associative map). Therefore I either had to start each method by transforming the morphism to an integer alphabet (which would be "clumsy"), or all the boolean arrays of size n indexed by letters I made use of in pseudocode had to be transformed into something different. The Python built-in container `set`, which represents the abstract data type of, well, a set, was for used for this purpose.

This includes the arrays R from Reach, M from MortalLetters and *new* from SimplifyNoninjective (which was also combined with the stack *todo*,

as `set`s also support the operation of popping an arbitrary element). Furthemore, both the linked list structures from MortalLetters were implemented using `set`s, as we do not actually require order and multiplicity in the algorithm.

This leads to some changes in the time complexity. The most popular Python implementation, called CPython, implements `set`s using a hash table. Therefore, a containment check has time complexity $O(1)$ only in the average case (and $O(n)$ in the worst case), while checking the i-th index of an array has time complexity $O(1)$ in all the cases. This is however not a concern in practice, as the overhead of Python is large enough such that accessing a hash table instead of accessing an array makes a negligible difference. Besides, using an array in Python (efficiently) requires calling C code, which is not done in SageMath without good enough reason.

Finally, I will discuss some design considerations that went into the methods for finding infinite periodic factors. In the bounded case an injective simplification is also obtained to make sure I am working with an injective morphism, even though this is not strictly necessary. However, it leads to some code being simplified: firstly, since in an injective morphism each letter is immortal, the check of whether there is at least one immortal label in a cycle turns into a check whether is at least one nonempty label in a cycle (this also allowed me to put the algorithm for mortal letters in a separate ticket). Secondly, it can be seen that without mortal letters each bounded word has the "delay" before it starts repeating (s_1) equal to zero, thus s_k is also always zero, and it can be ignored.

Another change from the pseudocode (for both the unbounded and bounded case) is that instead of returning all the conjugates of each primitive word v, such that v^ω is an infinite periodic factor, I only return the lexicographically minimal primitive conjugate from each equivalency class $[v^\omega]$. In other words from each equivalency class $[v^\omega]$ I return one representative v, such that $\forall u^\omega \in [v^\omega] : |v| \leq |u| \wedge v \leq u$. It can be seen that such a word v must be unique. For example instead of $\{abcd, bcda, cdab, dabc\}$ only $\{abcd\}$ is returned.

The reasoning for this is that this makes the output of these methods "cleaner" (for example in examples in the documentation), and if the bigger set with all the conjugates is desired, it can be computed from this smaller one more efficiently then the smaller one can be computed from the bigger one. In the pseudocode this could be accomplished by replacing the lines:

27: **for** for each conjugate c of v **do**
28: add c to V

---

with the line:

27: add MinimalConjugate(v) to V

The downside is that in SageMath there is not a method for finding the minimal conjugate of a word, so I had to implement my own (`minimal_conjugate`). The naive algorithm compares a word on input with all its conjugations, which leads to quadratic time complexity. There are multiple more sophisticated algorithms, which have linear time complexity. As they are nontrivial and nonessential to this thesis, I will omit their description. I implemented the one introduced by Jean Pierre Duval in his paper published in 1983 [36]. As the majority of the work in the algorithm is done by the so-called Lyndon factorization, which was already implemented in SageMath (`FiniteWord.lyndon_factorization`), the resulting code is quite short.

The implementation of the `is_pushy` and `is_unboundedly_repetitive` methods is not as efficient in the best-case as it could be, as we only need to find a single (un)bounded infinite periodic factor (or even we only need to just confirm some exists) to ascertain their truthiness, but the underlying methods only return after finding all of them.

This could be solved by providing a different implementation for these methods, which would only decide the existence of at least one infinite periodic factor. This would be very helpful for pushiness (as mentioned in the previous chapter, it would have linear time complexity instead of an uncertain one) and slightly helpful for unbounded repetitiveness. On the other hand, it would cause a nontrivial amount of code redundancy. The code could be deduplicated in a shared method, but methods that do not make much sense "on its own" are not a very popular practice in SageMath. For this reason, I decided to keep the less efficient version.

Before I started working on these methods I thought would solve this by making the underlying methods return Python generators instead of Python sets. Python generators are a sort of syntactic sugar that allows one to easily write methods that lazily return a sequence of values, that is the work required for returning the next value is deferred until the value is explicitly asked for. Then the methods `is_pushy` and `is_unboundedly_repetitive` could simply only ask for the first infinite periodic factor and never ask for the rest. However, here are the reasons why I didn't implement it like that in the end:

- The number of infinite periodic factors in "practical" morphisms is usually near zero, and everything is fast when it is small.

- It does not fully solve the problem for pushiness (we only need to find a cycle with an immortal label, not the corresponding bounded periodic factor).
- We still need to keep a set around, to keep us from returning duplicates, as the algorithms might find the same word multiple times.
- The main reason: It makes the implementation, documentation and usage more complicated.

Moving on, in the pseudocode for the algorithms for finding infinite periodic factors the axiom is only used at the beginning, and that is for reducing the D0L system. Intuitively, this is because the axiom is finite, but we are looking for infinite words. For a direct proof of the unimportance of the axiom see [5] (Proposition 3.2). This means that since the methods were implemented in the `WordMorphism` class, and not in a D0L system class, I could have omitted the axiom parameter and just assumed the morphism belongs to a reduced D0L system. However, I left the axiom parameter in, as I felt it can be useful, and that it makes the methods easier to understand.

Finally, a slight annoyance is caused by the fact that in some methods I made use of the `is_endomorphism` method from SageMath, which checks whether the domain and codomain are equal. However, the implemented methods work fine even if the codomain is a proper subset of the domain – this causes one to sometimes have to manually specify the codomain's alphabet. Nonetheless, as this is the case even for some other methods in SageMath, I left it this way for the time being.

3.4 Testing

The methods were tested not only with a few handpicked morphisms (such as the ones from the examples in the documentation and/or the ones from the examples in this text), but also with thousands of randomly generated ones.

The morphisms were randomly generated by first choosing the size of the alphabet uniformly between two numbers (the particular values were different for each method and will be omitted here for brevity), then choosing the sizes of the images using a modified geometric distribution (with a maximal limit) and finally by filling out the images with letters uniformly picked from the alphabet.

The problem with randomly generated morphisms is that the wanted results of the methods are unknown. That is why they either have to be checked against different implementations or checked only incompletely (see below).

For `reach` I compared the results with a slower naive algorithm, which simply iterates the axiom n times under the morphism (where n is the size of the alphabet) – it can be seen that this must uncover all the reachable letters.

For `growing_letters` (and therefore also indirectly for `mortal_letters`) I compared the results with the implementation already in SageMath.

For `is_injective` I compared the results with a modified version of the algorithm, which in the case of noninjectivity also returned a pair of conflicting words, which served as a certificate of noninjectivity. However, no such certificate was checked in the injective case.

If `simplify` returned a simplification, I tested that its definition holds. If it did not, I at least tested that the morphism satisfies two necessary conditions for elementary morphisms: injectivity and the one described here [15] (Theorem 1.2). For `simplify_injective` I simply tested that the returned result satisfies the definitions.

For `infinite_repetitions` I generated the set of factors of the language of the system up to a certain length (20) and then checked that the primitive roots of the found infinite repetitions and a limited number of their powers are in this set.

Even though these tests are only partial (for example the last one does not test in any way that these are indeed *all* the infinite repetitions), I still ended up finding a few bugs with them, which would otherwise possibly not have been uncovered.

I included in this thesis a file `test.py` containing these tests. It can be loaded in the same way as the `impl.py` file and the tests can be then run with the various `test_*` methods.

Chapter 4

Injective D0L systems

Definition. A D0L system $G = (\varphi, s)$ is *injective* if for all words u and v from $FL(G)$ we have, if $\varphi(u) = \varphi(v)$, then $u = v$.

In other words, a D0L system is injective, if its morphism is injective on (when its domain is restricted to) the set of the factors of the language of the system.

Clearly, if a D0L system has an injective morphism, then it also must be injective, but the converse is not true. For example (taken from [13]), see the D0L system $H = (\psi, a)$ with $\psi = \{a \mapsto abca, b \mapsto bca, c \mapsto a\}$. We have that ψ is noninjective, since $\psi(a) = \psi(cb)$, but it can be seen that cb is not in $FL(H)$ and that H is injective.

As was mentioned previously, it is not known whether the problem of deciding D0L system injectivity is decidable. The motivation behind the problem of deciding D0L system injectivity comes from its close relation to the problem of deciding D0L system circularity, which takes inspiration from coding theory.

Informally, a code is *circular* if any message written "on a circle" can be uniquely decoded [6]. For example, the code $\{a \mapsto 01, b \mapsto 10\}$ is uniquely decodable since no codeword is a prefix of each other. However, if we receive a message on a circle (see Figure 4.1) containing codewords of only one source symbol, then there are two possible decodings ($aa \dots a$ and $bb \dots b$).

A D0L system is *circular* if any sufficiently large word from the set of factors of the language of the system written on a circle "has at most one possible way a

[6] This informal definition is slightly misleading, as for example the code $a \mapsto 00$ is not circular – it also has to be known with certainty at what *position* on the circle to start the decoding.

Figure 4.1: Message on a circle.

preimage can be written on a circle". I will omit here the formal definitions of a circular code and a circular D0L system (which can be found, for example, in [42] and [13]), as they are nontrivial and not particularly required for this thesis.

Circular D0L systems form a useful subclass of D0L systems (they are used for example in [12] and [37]), but it is also not known whether the problem of deciding D0L system circularity is decidable. The following theorem proven by K. Klouda and Š. Starosta in their paper published in 2019 [13] (strengthening a theorem proven by Filippo Mignosi and Patrice Séébold in their paper published in 1993 [38]) gives a characterization of circular D0L systems:

Theorem 4.1 (K. Klouda and Š. Starosta [13] – Theorem 12)**.** *An injective D0L system is circular if and only if it is not unboundedly repetitive.*

This theorem, together with the fact that unbounded repetitiveness is decidable for D0L systems and that circularity for D0L systems is defined (mostly [13]) only for injective D0L systems, tells us that if the problem of deciding injectivity of a D0L system is decidable, then the problem of deciding circularity of a D0L system is also decidable.

We can simplify our work slightly by considering the following definition:

Definition. A D0L system $G = (\varphi, s)$ is *propagating* (denoted by PD0L) if φ is non-erasing.

Same as in the problem of injectivity of a morphism, if a D0L system has a (reachable) letter with an empty image, it is trivially noninjective. Since this is easily checkable, we can restrict ourselves to PD0L systems.

The rest of this chapter is structured as follows: first, it is shown that checking whether a word is a factor of the language of a PD0L system is doable in polynomial time, then it is shown that there is no known bound for the size of the "conflicting" factors, and finally, it is attempted to sidestep this issue by formulating the problem using formal languages.

4.1 "Factorship" problem for PD0L systems

Y. Kobayashi and F. Otto have proved the following [5] (Proposition 2.3 (b)):

Proposition 4.2. *Let G be a D0L system. Then $FL(G)$ is a context-sensitive language.*

The algorithm in this section is a simpler rendition of the approach in their proof. Nonetheless, thanks to it I have proved the following:

Theorem 4.3. *Let $G = (\varphi, s)$ be a PD0L system. Then deciding whether a word x is in $FL(G)$ can be done in polynomial time with regards to the size of G and x.*

Proof. Let the set of all the factors of $L(G)$ of length equal to an integer $k \geq 1$ be denoted by $FL_k(G)$. The algorithm works by generating the set $\bigcup_{k=1}^{|x|} FL_k(G)$ and then checking whether it contains x. The generation of factors is described in pseudocode in Algorithm 4.1. First, we must prove the algorithm's correctness.

Algorithm 4.1 FactorLanguage

Input: PD0L system G (morphism φ with letters $1 \dots n$ and axiom s) and an integer $k \geq 1$
Output: set F, containing all the factors of $L(G)$ of length less or equal to k

```
1: F ← an empty set
2: todo ← an empty stack
3: add the letter S (n + 1) to the morphism, with its image equal to s
4: push S on top of todo
5: while todo is not empty do
6:     u ← pop from todo
7:     v ← φ(u)
8:     for for each factor f of v of length less or equal to k do
9:         if f is not in F then
10:            add f to F
11:            push f on top of todo
12: return F
```

Lemma 4.4. *With a PD0L system G and an integer $k \geq 1$ on input, the output of the algorithm in Algorithm 4.1 is the set $\bigcup_{k=1}^{|x|} FL_k(G)$.*

Proof. We will prove it easily by induction on k.

Base case: $k = 1$. Then the algorithm is equal to the algorithm in Algorithm 1.1.

Inductive case: Let us say that the algorithm has already generated all the factors of $L(G)$ of length less or equal to k and we are proving whether an arbitrary factor v of $L(G)$ of length $k+1$ will also be generated. Clearly, since v is a factor of L, there must be a factor u of L such that v is a factor of $\varphi(u)$. Since G is propagating, the shortest such u must have a length of at most $k+1$.

If it is of length less than $k+1$, v will be generated from u, when u is popped from the stack, hence we are done, so let us say its length is equal to $k+1$. However, the word u must also have such a "predecessor". Following this chain of predecessors, we must either reach a factor of $L(G)$ of length less or equal to $k+1$, or we must reach the axiom, which is the first image processed by the algorithm, so v also has to be eventually generated. □

Here is an example of why the D0L system must be propagating. Let $H = (\psi, s)$ be a D0L system with $\psi = \{s \mapsto abc, a \mapsto a, b \mapsto \varepsilon, c \mapsto c\}$. Then $L(H) = \{s, abc, ac\}$, $FL_1(H) = \{s, a, b, c\}$, $FL_2(H) = \{ab, bc, ac\}$ and $FL_3(H) = \{abc\}$, but the smallest factor u of $L(H)$ such that the word ac is a factor of $\varphi(u)$ is the word abc.

Now, for the time complexity. On each popped word u we use φ once and then search its image for yet unseen factors of $L(G)$ of length less or equal to k. Clearly, we can do that in polynomial time with regards to k, the size of u, and the size of φ. This brings us to our last question, which is how many factors of $L(G)$ are there? This question was partially answered by A. Ehrenfeucht, Kwok Pun Lee and G. Rozenberg in their paper published in 1975 [39]:

Theorem 4.5 (A. Ehrenfeucht, Kwok Pun Lee and G. Rozenberg [39] – Theorem 2)**.** *Let G be a D0L system. The corresponding function* [7] *f_G from $\mathbb{N}^+$ to $\mathbb{N}$, defined as $f_G(n) = |FL_n(G)|$, is in $O(n^2)$.*

This theorem tells us that the size of the set $\bigcup_{k=1}^{|x|} FL_k(G)$ is polynomial with regards to the size of x. □

The algorithm is implemented in SageMath (`WordMorphism._language_naive`). It should be noted that whilst the the algorithm is polynomial, it is quite slow

[7] Such a function is usually called the factor complexity function and its study is of high importance in combinatorics on words (for an overview see [14].)

even on "practically" sized morphisms, as the exponent of the polynom is large. As a rough estimate, $|\bigcup_{k=1}^{|x|} FL_k(G)| = O(|x|^3)$, the size of an image of a word u is $O(|u|l)$ (recall from Chapter 1 that l is the size of the maximal image of φ), and a word v has $O(|v|^2)$ factors (there are $|v|$ positions for both their starting and their ending position), hence we have time complexity $O(|x|^3 \times (|x|l)^2 + (|s|l)^2 = |x|^5 \times l^2 + |s|^2 \times l^2)$.

A simple optimization opportunity is hidden in the fact, that we are when are finding the factors of $\varphi(u)$ for some factor $u = u_1u_2 \dots u_n$ (where $|u| \geq 2$), it can be seen that we only have to take into account factors which have at least one letter from $\varphi(u_1)$ and at least one letter from $\varphi(u_n)$, all the other factors of $\varphi(u)$ will surely be found when we pop the various factors of u from the stack – and all of these have to be popped at some point from the stack since u is also a factor.

This modification is described in pseudocode in Algorithm 4.2. As we now only have at most l possible positions for both the starting and ending positions of factors, the time complexity is improved to $O(|x|^3 \times l^2 + |s|^2)$. After some experimental testing, it was shown to be also notable in practice, hence the modification was submitted as a patch to SageMath – here is the corresponding ticket: [40].

4.2 The obvious approach

Now, that we have an algorithm for deciding whether a word x is in $FL(G)$ for some PD0L system G, it "should" be trivial to combine it with the algorithm for deciding injectivity of a morphism. The resulting algorithm would still have polynomial time complexity. However, we quickly run into a nontrivial issue.

Let us illuminate it with some notation. Let φ be a noninjective morphism. Then $short(\varphi)$ denotes the minimal sum of the length of two words u and v, such that $u \neq v$ and $\varphi(u) = \varphi(v)$. Let $G = (\varphi, s)$ be a noninjective D0L system. Then $short(G)$ denotes $short(\varphi)$, with the added restriction that u and v are in $FL(G)$.

We have the following lemma:

Lemma 4.6. *Let φ be a non-injective morphism. Then $short(\varphi) \leq m$.*

Proof. We need the concept of tails from the algorithm for deciding the injectivity of a morphism. We know that the number of tails in φ is bounded by m. If we keep track of pairs of preimages instead of tails in the algorithm, it

Algorithm 4.2 FactorLanguage (improved)

Input: PD0L system G (morphism φ with letters $1 \dots n$ and axiom s) and an integer $k \geq 1$

Output: set F, containing all the factors of $L(G)$ of length less or equal to k

1: $F \leftarrow$ an empty set
2: *todo* $\leftarrow$ an empty stack
3: add the letter S $(n+1)$ to the morphism, with its image equal to s
4: push S on top of *todo*
5: **while** *todo* is not empty **do**
6: $u \leftarrow$ pop from *todo*
7: $v \leftarrow \varphi(u)$
8: **if** $|u| = 1$ **then**
9: **for** $i \leftarrow 1$ to $|v|$ **do**
10: **for** $j \leftarrow i$ to $\text{Min}(i + k - 1, |v|)$ **do**
11: $f \leftarrow$ a factor of v from indices i to j (inclusive)
12: **if** f is not in F **then**
13: add f to F
14: push f on top of *todo*
15: **else**
16: *left* $\leftarrow |\varphi[\text{first letter of } u]|$
17: *right* $\leftarrow |\varphi[\text{last letter of } u]|$
18: *mid* $\leftarrow |v| -$ *left* $-$ *right*
19: $q \leftarrow k -$ *mid*
20: **for** $i \leftarrow \text{Max}(1,$ *left* $- q + 2)$ to *left* **do**
21: **for** $j \leftarrow$ *left* $+$ *mid* $+ 1$ to *left* $+$ *mid* $+ \text{Min}(q -$ *left* $+ i,$ *right*$)$ **do**
22: $f \leftarrow$ a factor of v from indices i to j (inclusive)
23: **if** f is not in F **then**
24: add f to F
25: push f on top of *todo*
26: **return** F

can be seen, that each letter added to either u or v creates a new tail (recall Figure 2.5).

Furthermore, each new tail must be different from all the previous ones, otherwise u and v would not be minimal. Proof by contradiction: Let u and v be words "with a repeated tail" satisfying the definition of non-injectivity of φ and $short(\varphi) = |u| + |v|$.

That is formally let $u_1, u_2, u_3, v_1, v_2, v_3, t$ be nonempty words such that $\varphi(u_1) = \varphi(v_1)t$ and $\varphi(u_1u_2) = \varphi(v_1v_2)t$ (see Figure 4.2 (Case 1)) or $\varphi(u_1) = \varphi(v_1)t$ and $\varphi(u_1u_2)t = \varphi(v_1v_2)$ (Case 2) or $\varphi(u_1)t = \varphi(v_1)$ and $\varphi(u_1u_2) = \varphi(v_1v_2)t$

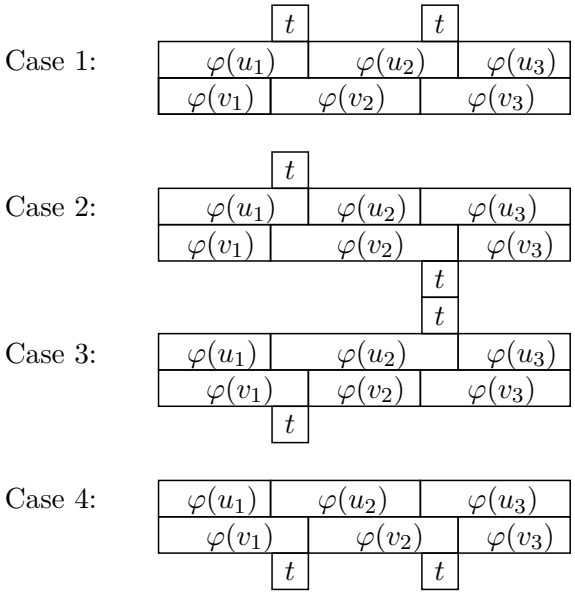

Figure 4.2: Auxiliary diagrams for a proof.

(Case 3) or $\varphi(u_1)t = \varphi(v_1)$ and $\varphi(u_1u_2)t = \varphi(v_1v_2)$ (Case 4) and finally let $u = u_1u_2u_3$ and $v = v_1v_2v_3$ be words such that $u \neq v$, $\varphi(u) = \varphi(v)$ and $short(\varphi) = |u| + |v|$.

Then it can be seen that we can obtain strictly shorter words $u' = u_1u_3$ and $v' = v_1v_3$ for Cases 1 & 4 and $u' = u_1v_3$ and $v' = v_1u_3$ for Cases 2 & 3, which do not have a repeated tail but still satisfy the definition of noninjectivity of φ, which is a contradiction with $short(\varphi) = |u| + |v|$.

Since tails can not repeat, we have that $short(\varphi)$ is bounded by the number of tails in φ, that is m. □

However, the above proof does *not* work for $short(G) \leq m$, where G is a noninjective D0L system, since u' and v' might not be in $FL(G)$. See for example $H = (\psi, adbc)$ with $\psi = a \mapsto a, b \mapsto ab, c \mapsto b, d \mapsto bb$. The words u and v are created as follows:

- $\psi(b)/\psi() = \text{ab}/_{\sqcup}$ – First tail is ab.
- $\psi(b)/\psi(a) = \text{ab/a}_{\sqcup}$ – Second tail is b, and $c \mapsto b$, but we cannot append c after a, since ac is not in $FL(H)$. However, $d \mapsto bb$ and ad is in $FL(H)$.
- $\psi(b)/\psi(ad) = \text{ab}_{\sqcup}/\text{abb}$ – Third tail is again b, but here we can append c after b, since bc is in $FL(H)$.

$$H_i = (\psi,\, a_0d_0c_0b_0d_0)$$

$a_0 \mapsto a_1$	$b_0 \mapsto b_1$	$c_0 \mapsto c_1$	$d_0 \mapsto d_1d_1$
		$\vdots$	
$a_j \mapsto a_{j+1}$	$b_j \mapsto b_{j+1}$	$c_j \mapsto c_{j+1}$	$d_j \mapsto d_{j+1}d_{j+1}$
		$\vdots$	
$a_i \mapsto a$	$b_i \mapsto ab$	$c_i \mapsto b$	$d_i \mapsto bb$
$a \mapsto a$	$b \mapsto b$		

Figure 4.3: D0L system used in a proof.

- $\psi(bc)/\psi(ad) = \text{abb/abb}$ – We found $u = bc$ and $v = ad$.

It can be seen, that these two words are the only ones that satisfy the definition of noninjectivity of H and whose sum of lengths is equal to $short(H)$. Hence, in D0L system injectivity it is sometimes required for a tail to repeat.

This gives rise to two issues:

1. In the algorithm for deciding injectivity we have to keep track of pairs of preimages instead of tails. At this point, since the algorithm for deciding whether x is in $FL(G)$ already generates all the factors of length less or equal to $|x|$, it is simpler to just expand the algorithm for deciding whether x is in $FL(G)$ to also check whether the images of all the factors are unique.
2. And more importantly, without a bound for $short(G)$, we do not know when to stop the algorithm.

It might seem possible that $short(G) \leq m$ is still true, just that it has to be proven in some other way. However, it turns out that not only is there a D0L system $H = (\psi, s)$ such that $short(H) > m$, but that also the following theorem holds.

Theorem 4.7. *There is an infinite sequence of D0L systems $(H_i)_{n\geq 0}$, such that their size grows linearly but $short(H_i)$ grows exponentially.*

Proof. The D0L system H_i is described in Figure 4.3. Whenever we increase i by 1, we increase n by 4, m by 5 and l and $|s|$ by 0. For the purposes of this proof let the size of a D0L system G be denoted by $|G|$ and computed with the equation: $n \times l + |s|$. Then we have $|H_i| = (6+4i)2+4 = 16+8i$. Clearly, $|H_i| = O(i)$.

Now, let us compute $short(H_i)$. For starters, here is $S(H_i)$:

$$\begin{aligned}
\varphi^0(s) &= a_0 d_0 c_0 b_0 d_0 \\
\varphi^1(s) &= a_1 d_1 d_1 c_1 b_1 d_1 d_1 \\
&\vdots \\
\varphi^i(s) &= a_i d_i^{2^i} c_i b_i d_i^{2^i} \\
\varphi^{i+1}(s) &= \left(a b^{2^{i+1}+1}\right)^2 \\
\varphi^{i+2}(s) &= \left(a b^{2^{i+1}+1}\right)^2 \\
&\vdots
\end{aligned}$$

It can be seen, that the only pair of words (u, v) satisfying the definition of noninjectivity of H_i is $(a_i d_i^{2^i} c_i,\ b_i d_i^{2^i})$. Then we have $short(H_i) = 3 + 2^{i+1}$. Clearly, $|H_i| = O\left(2^i\right)$. □

As a corollary, the algorithm for deciding injectivity of some D0L system G, that works by generating all the factors of $L(G)$ shorter than $short(G)$ and checking the uniqueness of their images, will run in at least exponential time with regards to the size of G.

4.3 A different approach

Let $G = (A, \varphi, s)$ be a D0L system. In this section, it is attempted to find an algorithm for deciding whether G is injective while avoiding the issue of finding an upper bound for $short(G)$, by representing the pairs of words (u, v) satisfying the definition of non-injectivity of φ and the pairs of words (e, f) from the language $FL(G)$ as formal languages, and then checking whether their intersection is nonempty.

We need some way to represent pairs of words (u, v) as one word w. Let / denote any one letter not in A, which will be used as a delimiter. For example, the pair of words (u, v) can be represented as one word w by simply concatenating them with a delimiter, that is $w = u/v$. Loosely speaking, in this section it is shown that there exists a representation, such that the language of pairs of words (u, v) satisfying the definition of non-injectivity φ is regular, and that there exists a representation, such that the language of pairs of words (e, f) from the language $FL(G)$ is extended non-deterministic Lindenmayer (abbreviated as E0L) (which will be defined later in the section).

This is notable because E0L languages are closed under intersection with regular languages [15] (Theorem 1.8) and their emptiness is decidable [41] (Theorem 19). However, the two representations mentioned in the previous

paragraph are incompatible, thus this does not lead us to a successful algorithm. Nonetheless, I found the propositions in this section interesting enough to warrant their inclusion in this thesis.

Let us start with the pairs of words (u, v) satisfying the definition of noninjectivity of φ. The representation will make use of the close relation between u and v. Let me repeat what was said at the start of the proof of Lemma 4.6: If the algorithm for deciding injectivity of a morphism tracks pairs of preimages (u, v) instead of tails, it can be seen, that each letter added to either u or v creates a new tail. Moreover, only the preimage with the currently smaller image can have a letter added. Hence the pair of words (u, v) can be uniquely represented by one word w without delimiters by mimicking the way that they would have been processed by the algorithm for deciding the injectivity of a morphism.

Let me show an example before the formal definition. Let $\psi = \{a \mapsto 11, b \mapsto 12, c \mapsto 123, d \mapsto 31112\}$ be a morphism (from Figure 2.5), with a pair of words $(u, v) = (cab, bd)$ satisfying the definition of noninjectivity of ψ. Let $\overline{u}$ and $\overline{v}$ denote the already processed parts of u and v. Then we have:

- $w = cb$ – We start with both the first letters of u and v to create the first tail. Since $|\psi(\overline{u} = c)| > |\psi(\overline{v} = b|)$, we continue with the second letter of v.
- $w = cbd$ – Since $|\psi(\overline{u} = c)| < |\psi(\overline{u} = bd|)$, we continue with the second letter of u.
- $w = cbda$ – Since $|\psi(\overline{u} = ca)| < |\psi(\overline{u} = bd|)$, we continue with the third letter of u.
- $w = cbdab$ – We are done.

Note that if the pair on input was (v, u) instead of (u, v), the result would have been the same except for the first two letters being switched ($w = bcdab$).

Now, even though we do not need to use a delimiter, they still make the result more readable. As such, let there be a delimiter / in the word w whenever we switch between the letters of u and v. Then for the (u, v) above we have $w = c/bd/ab$ (and $w = b/c/d/ab$ for (v, u)). The formal definition is as follows:

Definition. Let φ be a nonerasing morphism and let us during this definition denote the operation of concatenation by multiplication. Then a word $w = w_1/w_2/\ldots/w_n$ is a *conflict* of φ, if:

$$a \mapsto 1 \quad b \mapsto 011 \quad c \mapsto 01110 \quad d \mapsto 1110 \quad e \mapsto 10011$$

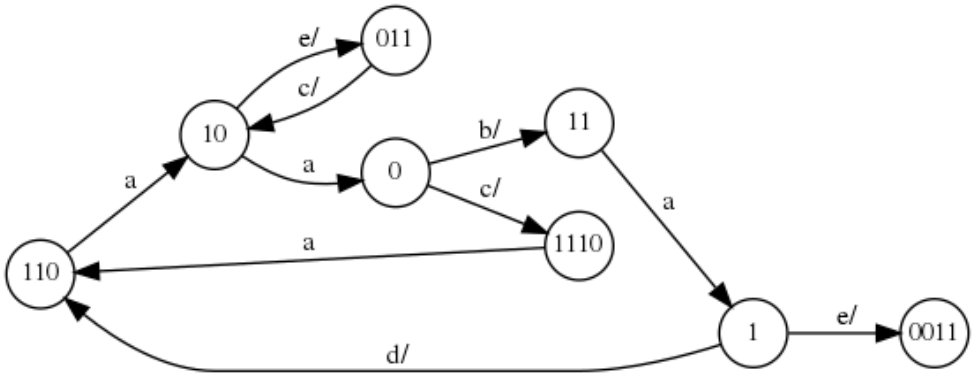

Figure 4.4: Graph of tails.

- there exist integers $n \geq 2$ and $m \geq 2$,
- a sequence of nonempty words $w_1, w_2, \ldots, w_n$ (let us denote the j-th letter of a word w_i by $w_{i,j}$)
- and a sequence of words $t_1, t_2, \ldots, t_m$ such that:
- $\sum_{i=1}^{n} |w_i| = m$, $|w_1| = 1$, $w_{1,1} \neq w_{2,1}$, $t_1 = \varphi(w_1)$, $t_m = \varepsilon$,
- for each integer $1 \leq j \leq m-1$ we have $t_j \neq \varepsilon$,
- for each integer $2 \leq i \leq n$ we have:

$$\prod_{k=1}^{\lceil i/2 \rceil} w_{2k-1} \times t_q = \prod_{k=1}^{\lfloor i/2 \rfloor} w_{2k}, \text{ where } q = \sum_{k=1}^{i} |w_i| \qquad \text{if } i \text{ is even,}$$

$$\prod_{k=1}^{\lfloor i/2 \rfloor} w_{2k} \times t_q = \prod_{k=1}^{\lceil i/2 \rceil} w_{2k-1}, \text{ where } q = \sum_{k=1}^{i} |w_i| \qquad \text{if } i \text{ is odd,}$$

- and for each integer $2 \leq i \leq n$ and for each integer $1 \leq j \leq |w_i|$ we have:

$$\prod_{k=1}^{\lfloor (i-1)/2 \rfloor} w_{2k} \times \prod_{k=1}^{j} w_{i,k} \times t_q = \prod_{k=1}^{\lceil i/2 \rceil} w_{2k-1}, \text{ where } q = \sum_{k=1}^{i-1} |w_i| + j \quad \text{if } i \text{ is even,}$$

$$\prod_{k=1}^{\lceil (i-1)/2 \rceil} w_{2k-1} \times \prod_{k=1}^{j} w_{i,k} \times t_q = \prod_{k=1}^{\lfloor i/2 \rfloor} w_{2k}, \text{ where } q = \sum_{k=1}^{i-1} |w_i| + j \quad \text{if } i \text{ is odd.}$$

Now, we can move on to the related proposition:

$a \mapsto 1 \quad b \mapsto 011 \quad c \mapsto 01110 \quad d \mapsto 1110 \quad e \mapsto 10011$

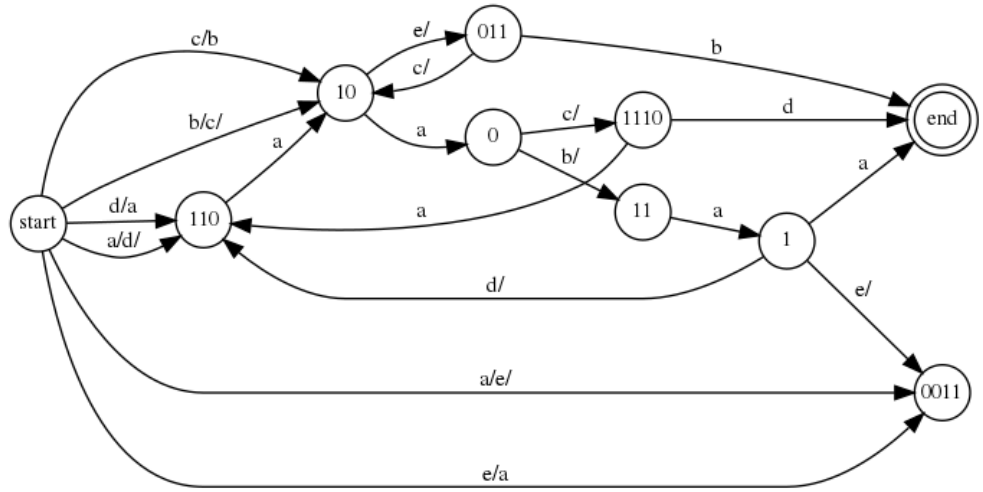

Figure 4.5: Finite automaton for a language of conflicts.

$a \mapsto 1 \quad b \mapsto 011 \quad c \mapsto 01110 \quad d \mapsto 1110 \quad e \mapsto 10011$

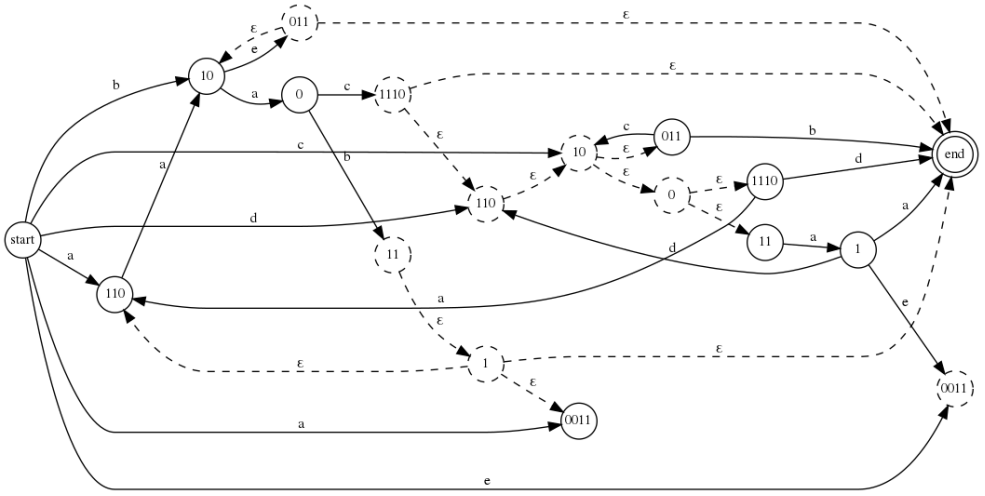

Figure 4.6: Finite automaton for minimal words with indistinct images. The duplicated states and transitions are represented by dashed lines.

Proposition 4.8. *Let φ be a nonerasing morphism. Then the language of conflicts of φ is regular.*

Proof. Let us begin by defining a graph of tails. A *graph of tails* of a morphism φ is a directed graph (with self-loops) (with labeled edges), where vertices are tails of φ and there is an edge from a vertex t_1 to a vertex t_2:

- with a label a, if $t_1 = \varphi(a)t_2$ for some letter a, or
- with a label $a/$, if $t_1\varphi(a) = t_2$ for some letter a.

For example, see Figure 4.4 (the morphism was taken from Wikipedia [23], where it was based upon [42] (Example 2.3.1)). The delimiter / again signifies when the order of the relative sizes of preimages is switched. In Section 2.2 it was proven that the Algorithm 2.5 finds all tails (unless it ends early), and the tails are even found using the same "rules" (line 18), which are used for the definition of the edges of the above graph, hence the algorithm can be easily modified to create this graph.

To turn this graph into a finite automaton recognizing conflicts, we only have to add an appropriate start and accepting state. Again, we will inspire ourselves using Algorithm 2.5. There is a transition from a state t to the accepting state *end* with an input a if $t = \varphi(a)$ for some letter a (line 16) and there is a transition from the start state *start* to a state t:

- with an input a/b, if $\varphi(a) = \varphi(b)t$ for some letters a and b, or
- with an input $a/b/$, if $\varphi(a)t = \varphi(b)$ for some letters a and b (line 9).

For example, see Figure 4.5. For the sake of clarity, multi-letter labels are used as a shorthand for sequences of nodes with single valid transitions and transitions which are not described in the visualisation are invalid (they lead to an omitted error state). As the Algorithm 2.5 is correct, automaton constructed this way for an arbitrary nonerasing morphism also has to be correct. □

We can use this proposition to prove two corollaries.

Corollary 4.9. *Let φ be a morphism. Then the language of words u, such that there exists a word v such that $u \neq v$ and $\varphi(u) = \varphi(v)$, is regular.*

Proof. If φ is erasing, then it can be seen that this language is equal to A^*, where A is the alphabet of the domain of φ, which is clearly a regular language. Let us therefore assume that φ is nonerasing.

First, let us describe how to obtain a subset of this language, where we also require for the conditions $\varphi(u_1) \neq \varphi(v_1)$ and $\varphi(u_m) \neq \varphi(v_n)$ to hold. To obtain this subset it suffices to remove all parts of one word and all delimiters from a conflict. This can be achieved by duplicating all the transitions and all the states (except the start and accepting ones) – the original states represent the first word and the duplicate states represent the second word.

Furthermore, all the transitions with inputs ending in a delimiter are changed (except for the ones from the start state, see the next paragraph) – the delimiter is omitted from the input and the target is changed to instead be the duplicate of the original target state (or the original, if the target was the duplicate one). Finally, all the transitions starting from duplicate states have their inputs changed to the empty words.

Special care is taken with the transitions from the start state. Transitions with inputs a/b for some letters a and b have their input changed to a and their target changed to the duplicate (as the conflict continues with the second word) and transitions with inputs $a/b/$ for some letters a and b have their inputs changed to a and their target unchanged (as the conflict continues with the first word).

For example, see Figure 4.6. The correctness of this automaton can be seen from the correctness of the last automaton. To also accept words where we have $u_1 = v_1$ or $v_m = v_n$, we only have to add the option of having arbitrary letters at the start or the end of the word – hence we just add a self-loop transition to the start state and the accepting state with the input being any letter. □

Corollary 4.10. *Let φ be a morphism. Then the language of words w, such that there exist words u and v such that $u \neq v$ and $\varphi(u) = w = \varphi(v)$, is regular.*

Proof. We simply replace the inputs in the last automaton with their images. □

Now, let us move on to the factors of a language of a D0L system. Y. Kobayashi and O. Friedrich have proven the following proposition:

Proposition 4.11 (Y. Kobayashi and O. Friedrich [5] – Proposition 2.1)**.**

(1) Let L be a regular language. Then $F(L)$ is a regular language.

(2) Let L be a context-free language. Then $F(L)$ is a context-free language.

As we have seen in Proposition 4.2, with D0L systems we only have the much weaker guarantee of context sensitivity. This is not very helpful, as many problems surrounding context-sensitive languages are undecidable (for example emptiness [41]). Hence, we need a step between context-free languages and context-sensitive languages. For that, we need a few more definitions regarding L-systems. These (and the examples too) are taken again from the book Mathematical Theory of L-systems [15]:

Definition. Let Σ be an alphabet. Then a *finite substitution* σ on Σ is a mapping from Σ to the set of subsets of Σ^*. Furthemore, if $v = v_1 v_2 \dots v_n$ is a word for some integer $n \geq 0$, then:

$$\sigma(w) = \{\varepsilon\} \qquad \text{if } w = \varepsilon,$$
$$\sigma(w) = \{w_1 w_2 \dots w_n | w_1 \in \sigma(v_1) \wedge w_2 \in \sigma(v_2) \wedge \dots \wedge w_n \in \sigma(v_n)\} \qquad \text{otherwise.}$$

Finally, if L is a set of words, then $\sigma(L) = \bigcup_{w \in L} \sigma(w)$.

For example, see $\Sigma = \{a\}$ and σ defined by $\sigma(a) = \{\varepsilon, a^2, a^7\}$. Then $\sigma(a^2) = \{\varepsilon, a^2, a^4, a^7, a^9, a^{14}\}$. Same as for morphisms, it is convenient to define finite substitutions by specifying the individual rules with the symbol $\mapsto$, that is the same finite substitution would be written as $\{a \mapsto \varepsilon, a \mapsto a^2, a \mapsto a^7\}$. Clearly, if $|\sigma(a)| = 1$ for all letters a from Σ, then it is equal to a morphism on words.

Definition. A *context-free Lindenmayer system* (abbreviated as an *0L system*) is a triple $G = (\Sigma, \sigma, \omega)$, where Σ is an alphabet, σ is a finite substitution on Σ and ω (referred to as the axiom) is an element of Σ^*. The language generated by an 0L system (denoted by $L(G)$) is equal to

$$L(G) = \{\omega\} \cup \sigma(\omega) \cup \sigma(\sigma(\omega)) \cup \dots = \bigcup_{i \geq 0} \sigma^i(\omega).$$

For example, see the 0L system $G = (\{a\}, \{a \mapsto a, a \mapsto aa\}, a\}$. Then $L(G) = \{a^n \mid n \geq 1\}$.

Definition. An *extended* context-free Lindenmayer system (abbrevaited as an *EOL system*) is a quadruple $G = (\Sigma, \sigma, \omega, \Delta)$, where $U = (\Sigma, \sigma, \omega)$ is the underlying 0L system and Δ (referred to as the terminal alphabet) is a subset of Σ. The language generated by an E0L system (denoted by $L(G)$) is equal to $L(U(G)) \cap \Delta^*$.

For example, see the E0L system $G = (\{S, a, b\}, \{S \mapsto a, S \mapsto b, a \mapsto a, a \mapsto aa, b \mapsto b, b \mapsto bb\}, \{a, b\})$. Then $L(G) = \{a^n \mid n \geq 1\} \cup \{b^n \mid n \geq 1\}$.

Now, we can prove the following proposition:

Proposition 4.12. *Let L be an E0L language. Then $F(L)$ is an E0L language.*

Proof. This will be proven by showing how to modify the E0L system generating L to make it generate $F(L)$ instead.

Let $G = (\Sigma, h, \omega, \Delta)$ be an E0L system generating L. Let us first show how to create an EOL system $G' = (\Sigma', h', \omega', \Delta')$ accepting all *prefixes* of words in $L(G)$. We start by "copying" over the alphabets and the rules, that is $\Sigma' = \Sigma$, $h' = h$ and $\Delta' = \Delta$. To simplify working with the axiom, we take some letter s not in Σ, add it to Σ' and add the rule $s \mapsto \omega$ to h'. Then for each letter a in $\Sigma \cup \{s\}$ we do the following:

- add the letter $\overrightarrow{a}$ to Σ', and if there is a rule $a \mapsto \varepsilon$ in h, we add the rule $\overrightarrow{a} \mapsto \varepsilon$ to h'
- and for each rule $a \mapsto v_1 v_2 \dots v_n$ $(n \geq 1)$ in h, where v_1, v_2, …, v_n are from Σ, we add the following rules to h':

$$\begin{aligned}
\overrightarrow{a} &\mapsto v_1 v_2 \dots v_{n-1} \overrightarrow{v_n} \\
\overrightarrow{a} &\mapsto v_1 v_2 \dots v_{n-1} v_n \\
\overrightarrow{a} &\mapsto v_1 v_2 \dots \overrightarrow{v_{n-1}} \\
\overrightarrow{a} &\mapsto v_1 v_2 \dots v_{n-1} \\
&\vdots \\
\overrightarrow{a} &\mapsto v_1 \overrightarrow{v_2} \\
\overrightarrow{a} &\mapsto v_1 v_2 \\
\overrightarrow{a} &\mapsto \overrightarrow{v_1} \\
\overrightarrow{a} &\mapsto v_1
\end{aligned}$$

Finally, we take another letter S not in Σ, add it to Σ', add rules $S \mapsto \varepsilon$, $S \mapsto s$, $S \mapsto \overrightarrow{s}$ to h and set $\omega' = S$.

For example, see Figure 4.7. Any letter with a right arrow accent can omit an arbitrary number of letters "from the right" in its rules, and at any time only the rightmost letter can have a right arrow accent – hence it can be seen that $L(G')$ is equal to the set of all prefixes of words in $L(G)$.

$$G = (\{a, b, c\}, h, abc, \{a, b, c\})$$
$$h = \{a \mapsto aa, b \mapsto bb, c \mapsto cc\}$$
$$G' = (\Sigma', h', S, \{a, b, c\})$$
$$\Sigma' = \{S, s, \overrightarrow{s}, a, \overrightarrow{a}, b, \overrightarrow{b}, c, \overrightarrow{c}\}$$
$$h'$$

$S \mapsto \varepsilon$	$S \mapsto s$	$S \mapsto \overrightarrow{s}$	
$s \mapsto abc$	$a \mapsto aa$	$b \mapsto bb$	$c \mapsto cc$
$\overrightarrow{s} \mapsto ab\overrightarrow{c}$	$\overrightarrow{s} \mapsto a\overrightarrow{b}$	$\overrightarrow{s} \mapsto \overrightarrow{a}$	
$\overrightarrow{s} \mapsto abc$	$\overrightarrow{s} \mapsto ab$	$\overrightarrow{s} \mapsto a$	
$\overrightarrow{a} \mapsto a\overrightarrow{a}$	$\overrightarrow{a} \mapsto \overrightarrow{a}$	$\overrightarrow{a} \mapsto aa$	$\overrightarrow{a} \mapsto a$
$\overrightarrow{b} \mapsto b\overrightarrow{b}$	$\overrightarrow{b} \mapsto \overrightarrow{b}$	$\overrightarrow{b} \mapsto bb$	$\overrightarrow{b} \mapsto b$
$\overrightarrow{c} \mapsto c\overrightarrow{c}$	$\overrightarrow{c} \mapsto \overrightarrow{c}$	$\overrightarrow{c} \mapsto cc$	$\overrightarrow{c} \mapsto c$

Figure 4.7: EOL system generating the set of all prefixes of an E0L language.

To augment this approach to create an E0L system $G'' = (\Sigma'', h'', \omega'', \Delta'')$ accepting all *factors* of words in $L(G)$, we have to create analogous letters with left arrows accents, that can omit an arbitrary number of letters from the left, and also slightly more powerful letters with bilateral arrow accents, that can omit an arbitrary number of letters from both the left and the right.

Formally: We set $\Sigma'' = \Sigma$, $h'' = h$ and $\Delta'' = \Delta$ and take some letter s not in Σ, add it to Σ'' and add a rule $s \mapsto \omega$ to h''. Then for each letter a in $\Sigma \cup \{s\}$ we do the following:

- add the letters $\overleftarrow{a}$, $\overrightarrow{a}$ and $\overleftrightarrow{a}$ to Σ''
- and if there is a rule $a \mapsto \varepsilon$ in h, we add the rules $\overrightarrow{a} \mapsto \varepsilon$, $\overleftarrow{a} \mapsto \varepsilon$ and $\overleftrightarrow{a} \mapsto \varepsilon$ to h''
- and for each rule $a \mapsto v_1 v_2 \dots v_n$ $(n \geq 1)$ in h, where v_1, v_2, …, v_n are from Σ, we do the following:
- for each integer $1 \leq i \leq n$ we add the following rules to h'':

$$\overleftarrow{a} \mapsto \overleftarrow{v_i} v_{i+1} \dots v_n$$
$$\overleftarrow{a} \mapsto v_i v_{i+1} \dots v_n$$
$$\overrightarrow{a} \mapsto v_1 \dots v_{i-1} \overrightarrow{v_i}$$
$$\overrightarrow{a} \mapsto v_1 \dots v_{i-1} v_i$$
$$\overleftrightarrow{a} \mapsto \overleftrightarrow{v_i}$$
$$\overleftrightarrow{a} \mapsto v_i$$

$$G = (\{a,b,c\}, h, abc, \{a,b,c\})$$
$$h = \{a \mapsto aa, b \mapsto bb, c \mapsto cc\}$$
$$G'' = (\Sigma'', h'', S, \{a,b,c\})$$
$$\Sigma'' = \{S, s, \overleftarrow{s}, \overrightarrow{s}, \overleftrightarrow{s}, a, \overleftarrow{a}, \overrightarrow{a}, \overleftrightarrow{a}, b, \overleftarrow{b}, \overrightarrow{b}, \overleftrightarrow{b}, c, \overleftarrow{c}, \overrightarrow{c}, \overleftrightarrow{c}\}$$
$$h''$$

$S \mapsto \varepsilon$			
$S \mapsto s$	$S \mapsto \overleftarrow{s}$	$S \mapsto \overrightarrow{s}$	$S \mapsto \overleftrightarrow{s}$
$s \mapsto abc$	$a \mapsto aa$	$b \mapsto bb$	$c \mapsto cc$
$\overleftarrow{s} \mapsto \overleftarrow{a}bc$	$\overleftarrow{s} \mapsto \overleftarrow{b}c$	$\overleftarrow{s} \mapsto \overleftarrow{c}$	
$\overleftarrow{s} \mapsto abc$	$\overleftarrow{s} \mapsto bc$	$\overleftarrow{s} \mapsto c$	
$\overrightarrow{s} \mapsto ab\overrightarrow{c}$	$\overrightarrow{s} \mapsto a\overrightarrow{b}$	$\overrightarrow{s} \mapsto \overrightarrow{a}$	
$\overrightarrow{s} \mapsto abc$	$\overrightarrow{s} \mapsto ab$	$\overrightarrow{s} \mapsto a$	
$\overleftrightarrow{s} \mapsto \overleftarrow{a}b\overrightarrow{c}$	$\overleftrightarrow{s} \mapsto \overleftarrow{a}\overrightarrow{b}$	$\overleftrightarrow{s} \mapsto \overleftarrow{b}\overrightarrow{c}$	
$\overleftrightarrow{s} \mapsto \overleftrightarrow{a}$	$\overleftrightarrow{s} \mapsto \overleftrightarrow{b}$	$\overleftrightarrow{s} \mapsto \overleftrightarrow{c}$	
$\overleftrightarrow{s} \mapsto abc$	$\overleftrightarrow{s} \mapsto ab$	$\overleftrightarrow{s} \mapsto bc$	
$\overleftrightarrow{s} \mapsto a$	$\overleftrightarrow{s} \mapsto b$	$\overleftrightarrow{s} \mapsto c$	
$\overleftarrow{a} \mapsto \overleftarrow{a}a$	$\overleftarrow{a} \mapsto \overleftarrow{a}$	$\overleftarrow{a} \mapsto aa$	$\overleftarrow{a} \mapsto a$
$\overrightarrow{a} \mapsto a\overrightarrow{a}$	$\overrightarrow{a} \mapsto \overrightarrow{a}$	$\overrightarrow{a} \mapsto aa$	$\overrightarrow{a} \mapsto a$
$\overleftrightarrow{a} \mapsto \overleftarrow{a}\overrightarrow{a}$	$\overleftrightarrow{a} \mapsto \overleftrightarrow{a}$	$\overleftrightarrow{a} \mapsto aa$	$\overleftrightarrow{a} \mapsto a$
$\overleftarrow{b} \mapsto \overleftarrow{b}b$	$\overleftarrow{b} \mapsto \overleftarrow{b}$	$\overleftarrow{b} \mapsto bb$	$\overleftarrow{b} \mapsto b$
$\overrightarrow{b} \mapsto b\overrightarrow{b}$	$\overrightarrow{b} \mapsto \overrightarrow{b}$	$\overrightarrow{b} \mapsto bb$	$\overrightarrow{b} \mapsto b$
$\overleftrightarrow{b} \mapsto \overleftarrow{b}\overrightarrow{b}$	$\overleftrightarrow{b} \mapsto \overleftrightarrow{b}$	$\overleftrightarrow{b} \mapsto bb$	$\overleftrightarrow{b} \mapsto b$
$\overleftarrow{c} \mapsto \overleftarrow{c}c$	$\overleftarrow{c} \mapsto \overleftarrow{c}$	$\overleftarrow{c} \mapsto cc$	$\overleftarrow{c} \mapsto c$
$\overrightarrow{c} \mapsto c\overrightarrow{c}$	$\overrightarrow{c} \mapsto \overrightarrow{c}$	$\overrightarrow{c} \mapsto cc$	$\overrightarrow{c} \mapsto c$
$\overleftrightarrow{c} \mapsto \overleftarrow{c}\overrightarrow{c}$	$\overleftrightarrow{c} \mapsto \overleftrightarrow{c}$	$\overleftrightarrow{c} \mapsto cc$	$\overleftrightarrow{c} \mapsto c$

Figure 4.8: EOL system generating the set of all factors of an E0L language.

- and for each integer $1 \leq i \leq n$ and for each integer $i \leq j \leq n$ we add the following rules to h'':

$$\overleftrightarrow{a} \mapsto \overleftarrow{v_i} v_{i+1} \dots v_{j-1} \overrightarrow{v_j}$$
$$\overleftrightarrow{a} \mapsto v_i v_{i+1} \dots v_{j-1} v_j$$

Finally, we take another letter S not in Σ, add it to Σ'', add rules $S \mapsto \varepsilon$, $S \mapsto s$, $S \mapsto \overleftarrow{s}$, $S \mapsto \overrightarrow{s}$ and $S \mapsto \overleftrightarrow{s}$ to h and set $\omega'' = S$.

For example, see Figure 4.8. Due to analogous reasons as for G', it can be seen that $L(G'')$ is equal to the set of all factors of $L(G)$. □

Wielding this proposition, we can obtain a representation of pairs of factors (e, f) by simply concatenating the language of factors with itself (and adding a delimiter).

Corollary 4.13. *Let L be a PD0L language. Then the language of words $w = e/f$, where $/$ is a delimiting symbol not in the alphabet of L and the words e and f are factors of L, is an E0L language.*

Proof. As PD0L languages are a subset of E0L languages, let $G = (\Sigma, h, \omega, \Delta)$ be an E0L system generating L and $G'' = (\Sigma'', h'', S, \Delta'')$ an E0L system generating $F(L)$ as per previous proposition. Then we simply add yet another letter Z not in Σ with the rule $Z \mapsto S/S$ and set it as the axiom. The only slight obstacle is caused by the fact, that each word from the pair might require a different number of iterations. This can be overcome by also adding the rule $S \mapsto S$. □

As was mentioned at the start of this section, E0L languages are closed under intersection with regular languages and deciding (given an E0L system) whether they generate the empty language is decidable. Unfortunately, it can be seen, that to solve the problem of D0L system injectivity we cannot simply intersect the languages from Proposition 4.8 and Corollary 4.13, as the former has a large amount of "permeation" between the two words in a pair, while the latter has none.

Furthermore, attempts to switch between these representations will also not work, as the language of pairs of words (u, v) without permeation (that is, represented as u/v) satisfying the definition of morphism noninjectivity is clearly not regular, as the word v is highly dependant upon the word u, and the language of pairs of factors (e, f) with permeation (that is, represented as $e_0e_1/f_0f_1f_2/\dots$) is intuitively not E0L, as its parsing would require running concurrently two "E0L automatons".

Finally, intersecting the languages from Corollary 4.9 (or even Corollary 4.10) and Proposition 4.12 will also not help us, as it tells us whether there is a factor which has two possible preimages, but it does not tell us, whether *both* of these preimages are also factors.

Conclusion

In the first part of this thesis, several sophisticated algorithms related to repetitive D0L systems were thoroughly explained, carefully analyzed, successfully implemented (using SageMath's combinatorics on words module), and methodically tested. The implementation was written from the start with the goal of contributing it into SageMath itself and it therefore follows SageMath's guidelines regarding coding practices and documentation writing.

This integration is at the time of writing an ongoing process. The additions were split into three patches [28][29][40]. Of these, one was already accepted, which means that it will be included in the next minor version of SageMath, while the other two are still under the process of review. It is the hope of the author that they will also be eventually accepted and moreover that these changes will facilitate further research into various problems in combinatorics on words, for example in the problem of finding bispecial factors in D0L systems [37] in the repetitive case.

In the second part of this thesis, a certain problem of unknown decidability related to circular D0L systems was discussed. While its decidability was not resolved, few nontrivial propositions were shown to be true, which the author hopes will be helpful in eventually settling this problem.

During the effort of writing this thesis, the author also reported and fixed several bugs in SageMath, some of which were trivial and some of which were nontrivial. The full list of both is available here in this query [43] in SageMath's ticketing system.

Bibliography

1. LOTHAIRE, Monsieur. *Combinatorics on Words*. 2nd ed. Cambridge University Press, 1997. Cambridge Mathematical Library. Available from DOI: 10.1017/CBO9780511566097.
2. LINDENMAYER, Aristid. Mathematical models for cellular interactions in development I. Filaments with one-sided inputs. *Journal of Theoretical Biology*. 1968, vol. 18, no. 3, pp. 280–299. ISSN 0022-5193. Available from DOI: 10.1016/0022-5193(68)90079-9.
3. LOTHAIRE, Monsieur. *Algebraic Combinatorics on Words*. Cambridge University Press, 2002. Encyclopedia of Mathematics and its Applications. Available from DOI: 10.1017/CBO9781107326019.
4. EHRENFEUCHT, Andrzej; ROZENBERG, Grzegorz. Repetition of subwords in DOL languages. *Information and Control*. 1983, vol. 59, no. 1, pp. 13–35. ISSN 0019-9958. Available from DOI: 10.1016/S0019-9958(83)80028-X.
5. KOBAYASHI, Yuji; OTTO, Friedrich. Repetitiveness of languages generated by morphisms. *Theoretical Computer Science*. 2000, vol. 240, no. 2, pp. 337–378. ISSN 0304-3975. Available from DOI: 10.1016/S0304-3975(99)00238-8.
6. KLOUDA, Karel; STAROSTA, Štěpán. An algorithm for enumerating all infinite repetitions in a D0L-system. *Journal of Discrete Algorithms*. 2015, vol. 33, pp. 130–138. ISSN 1570-8667. Available from DOI: 10.1016/j.jda.2015.03.006.
7. *SageMath* [online] [visited on 2020-11-01]. Available from: https://www.sagemath.org/.
8. GALLAGER, Robert Gray. *Information Theory and Reliable Communication*. Wiley, 1968. ISBN 9780471290483.

9. EVEN, Shimon. *Graph Algorithms*. Computer Science Press, 1979. ISBN 9780914894216.
10. EHRENFEUCHT, Andrzej; ROZENBERG, Grzegorz. Simplifications of homomorphisms. *Information and Control*. 1978, vol. 38, no. 3, pp. 298–309. ISSN 0019-9958. Available from DOI: 10.1016/S0019-9958(78)90095-5.
11. HARJU, Tero; KARHUMÄKI, Juhani. On the defect theorem and simplifiability. *Semigroup Forum*. 1986, vol. 33, no. 1, pp. 199–217. ISSN 1432-2137. Available from DOI: 10.1007/BF02573193.
12. CASSAIGNE, Julien. An algorithm to test if a given circular HDOL-language avoids a pattern. In: *in: IFIP World Computer Congress'94*. Elsevier (North-Holland, 1994, pp. 459–464.
13. KLOUDA, Karel; STAROSTA, Štěpán. Characterization of circular D0L-systems. *Theoretical Computer Science*. 2019, vol. 790, pp. 131–137. ISSN 0304-3975. Available from DOI: 10.1016/j.tcs.2019.04.021.
14. CASSAIGNE, Julien; NICOLAS, François. Factor complexity. In: *Combinatorics, Automata and Number Theory*. Ed. by BERTHÉ, Valérie; RIGO, Michel. Cambridge University Press, 2010, pp. 163–247. Encyclopedia of Mathematics and its Applications. Available from DOI: 10.1017/CBO9780511777653.005.
15. ROZENBERG, Grzegorz; ARTO, Salomaa. *The Mathematical Theory of L Systems*. Academic Press, 1980. ISBN 9780080874067.
16. HOPCROFT, John Edward; ULLMAN, Jeffrey David. *Introduction to Automata Theory, Languages and Computation*. 1st ed. Addison-Wesley, 1979. ISBN 9780201029888.
17. DIESTEL, Reinhard. *Graph Theory*. 5th ed. Springer, 2016. ISBN 9783662536216.
18. ALLOUCHE, Jean-Paul; SHALLIT, Jeffrey. The Ubiquitous Prouhet-Thue-Morse Sequence. In: DING, C.; HELLESETH, T.; NIEDERREITER, H. (eds.). *Sequences and their Applications*. London: Springer London, 1999, pp. 1–16. ISBN 978-1-4471-0551-0.
19. CORMEN, Thomas H.; LEISERSON, Charles Eric; RIVEST, Ronald Linn; STEIN, Clifford Seth. *Introduction to Algorithms*. 3rd ed. MIT Press, 2009. ISBN 9780262033848.
20. VITÁNYI, Paul M. B. On the size of D0L languages. In: *L Systems*. Ed. by ROZENBERG, Grzegorz; SALOMAA, Arto. Berlin, Heidelberg: Springer Berlin Heidelberg, 1974, pp. 78–92. ISBN 978-3-540-37823-5. Available from DOI: 10.1007/3-540-06867-8_6.
21. LANDO, Barbara. Periodicity and ultimate periodicity of D0L systems. *Theoretical Computer Science*. 1991, vol. 82, no. 1, pp. 19–33. ISSN 0304-3975. Available from DOI: 10.1016/0304-3975(91)90169-3.

22. WIKIPEDIA CONTRIBUTORS. *Pseudoforest — Wikipedia, The Free Encyclopedia* [https://en.wikipedia.org/w/index.php?title=Pseudoforest&oldid=990853275]. 2021. [Online; accessed 17-March-2021].

23. WIKIPEDIA CONTRIBUTORS. *Sardinas–Patterson algorithm — Wikipedia, The Free Encyclopedia*. 2021. Available also from: https://en.wikipedia.org/w/index.php?title=Sardinas%E2%80%93Patterson_algorithm&oldid=1014158965 [Online; accessed 3-April-2021].

24. RODEH, Michael. A fast test for unique decipherability based on suffix trees (Corresp.) *IEEE Transactions on Information Theory*. 1982, vol. 28, no. 4, pp. 648–651. Available from DOI: 10.1109/TIT.1982.1056535.

25. APOSTOLICO, Alberto; GIANCARLO, Raffaele. Pattern Matching Machine Implementation of a Fast Test for Unique Decipherability. *Inf. Process. Lett.* 1984, vol. 18, no. 3, pp. 155–158. ISSN 0020-0190. Available from DOI: 10.1016/0020-0190(84)90020-6.

26. HOFFMANN, Christoph M. A Note on Unique Decipherability. In: *Proceedings of the Mathematical Foundations of Computer Science 1984*. Berlin, Heidelberg: Springer-Verlag, 1984, pp. 50–63. ISBN 3540133720. Available from DOI: 10.1007/BFb0030289.

27. OEIS FOUNDATION INC. *The On-Line Encyclopedia of Integer Sequences* [https://oeis.org/A014062]. 2021.

28. *SageMath – Ticket #18119* [online] [visited on 2021-05-03]. Available from: https://trac.sagemath.org/ticket/18119.

29. *SageMath – Ticket #31684* [online] [visited on 2021-05-03]. Available from: https://trac.sagemath.org/ticket/31684.

30. *SageMathCell* [online] [visited on 2020-11-20]. Available from: https://sagecell.sagemath.org/.

31. *CoCalc* [online] [visited on 2020-11-01]. Available from: https://cocalc.com/.

32. *A page on SageMath's wiki* [online] [visited on 2020-11-01]. Available from: https://wiki.sagemath.org/days110.

33. *A question on Ask Sage: Geometric name of sage logo? Existence of dual?* [online] [visited on 2020-11-28]. Available from: http://ask.sagemath.org/question/7667/geometric-name-of-sage-logo-existence-of-dual/.

34. *Source code for the combinatorics on words module of SageMath (develop branch)* [online] [visited on 2020-11-28]. Available from: https://github.com/sagemath/sage/tree/develop/src/sage/combinat/words.

35. *Documentation for the combinatorics on words module of SageMath* [online] [visited on 2020-11-28]. Available from: https://doc.sagemath.org/html/en/reference/combinat/sage/combinat/words/__init__.html.

36. PIERRE DUVAL, Jean. Factorizing words over an ordered alphabet. *Journal of Algorithms*. 1983, vol. 4, no. 4, pp. 363–381. ISSN 0196-6774. Available from DOI: 10.1016/0196-6774(83)90017-2.

37. KLOUDA, Karel. Bispecial factors in circular non-pushy D0L languages. *Theoretical Computer Science*. 2012, vol. 445, pp. 63–74. ISSN 0304-3975. Available from DOI: https://doi.org/10.1016/j.tcs.2012.05.007.

38. MIGNOSI, Filippo; SÉÉBOLD, Patrice. If a DOL language is k-power free then it is circular. In: LINGAS, Andrzej; KARLSSON, Rolf; CARLSSON, Svante (eds.). *Automata, Languages and Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 507–518.

39. EHRENFEUCHT, Andrzej; LEE, Kwok Pun; ROZENBERG, Grzegorz. Subword complexities of various classes of deterministic developmental languages without interactions. *Theoretical Computer Science*. 1975, vol. 1, no. 1, pp. 59–75. ISSN 0304-3975. Available from DOI: 10.1016/0304-3975(75)90012-2.

40. *SageMath – Ticket #31760* [online] [visited on 2021-05-03]. Available from: https://trac.sagemath.org/ticket/31760.

41. JONES, Neil D.; SKYUM, Sven. Complexity of some problems concerning L systems. *Theory of Computing Systems*. 1979, vol. 13, pp. 29–43. ISSN 1432-4350. Available from DOI: 10.1007/BF01744286.

42. BERSTEL, Jean; PERRIN, Dominique; REUTENAUER, Christophe. *Codes and Automata*. Cambridge University Press, 2009. Encyclopedia of Mathematics and its Applications. Available from DOI: 10.1017/CBO9781139195768.

43. *SageMath – Tickets containing "Rejmon" in the author field* [online] [visited on 2021-05-03]. Available from: https://trac.sagemath.org/query?author=~Rejmon.

44. WIKIPEDIA CONTRIBUTORS. *L-system — Wikipedia, The Free Encyclopedia* [https://en.wikipedia.org/w/index.php?title=L-system&oldid=1018655516]. 2021. [Online; accessed 23-April-2021].

45. PRUSINKIEWICZ, Przemyslaw; LINDENMAYER, Aristid. *The Algorithmic Beauty of Plants*. Springer, 1990. ISBN 9780387946764.

Appendix A

Contents of enclosed CD

- `root`
  - `docs.pdf` ..... documentation created from `impl.py` using SageMath's documentation builder (Sphinx)
  - `impl.py` ..... Python source for the implementation
  - `latex/` ..... LaTeX source for this text
  - `test.py` ..... Python source for the tests
  - `thesis.pdf` ..... this text

Appendix B

Fractal plant

While L-systems generate strings, these strings can be turned into images using the so-called turtle graphics – the images are made by following the path of an imaginary turtle, which travels around a 2D plane (or in the more complicated variations, even a 3D space). Individual symbols from the alphabet of the string are given the meanings of movement commands and the turtle then carries them out sequentially based on the input string. L-systems come in by the way of being the natural vehicle for describing fractal-like imagery.

The particular image in the introduction was generated using the L-system extension in the graphics software Inkscape with the following D0L system (axiom = X) after 6 iterations:

$$\begin{aligned} \mathrm{X} &\mapsto \mathrm{F{+}[[X]{-}X]{-}F[{-}FX]{+}X} \\ \mathrm{F} &\mapsto \mathrm{FF} \\ + &\mapsto + \\ - &\mapsto - \\ [&\mapsto [\\ ] &\mapsto] \end{aligned}$$

where "X" has no meaning for the turtle, "F" means move forward one unit, "+" means turn left 25 degrees, "−" means turn right 25 degrees, "[" means push the current position and angle on a stack and "]" means pop the position and angle from the stack and restore it. This system was taken from the Wikipedia article on L-systems [44] (Example 7), where it was not sourced. The wikipedia article also contains more examples and even more information can be found, for example, in the book The Algorithmic Beauty of Plants by Przemyslaw Prusinkiewicz and A. Lindenmayer [45].