Insert here your thesis' task.

Czech Technical University in Prague
Faculty of Information Technology
Department of Theoretical Computer Science

Master's thesis

Algorithms generating bispecial factors in D0L-systems

Bc. Lenka Čačková

Supervisor: Ing. Štěpán Starosta, Ph.D

15th May 2014

Acknowledgements

I would like to thank my supervisor Ing. Štěpán Starosta, Ph.D for the patient guidance and advice he has provided.

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 particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 15th May 2014 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Čačková, Lenka. *Algorithms generating bispecial factors in D0L-systems.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2014.

Abstrakt

Táto diplomová práca pojednáva o non-pushy a cirkulárnych D0L-systémoch. Poskytujeme v nej prehl'ad hlavných výsledkov týkajúcich sa D0L-systémov, hlavne metódy popisujúcej všetky bišpeciálne faktory v pevnom bode morfizmu. Na základe analýzy tejto metódy predkladáme návrh a implementáciu jej algoritmu v algebra systéme SAGE.

Kl'účové slová bišpeciálne faktory, cirkulárne D0L-systémy, non-pushy D0L-systémy, pevné body morfizmov, Sage

Abstract

This thesis deals with non-pushy and circular D0L-systems. We present a summary of the known results on D0L-systems, especially concerning the method to describe all bispecial factors in a fixed point of a morphism. Based on analysis of this methods we design and implement its algorithm in computer algebra system SAGE.

Keywords bispecial factors, circular D0L-systems, non-pushy D0L-systems, fixed points of morphism, Sage

Contents

List of Figures

Introduction

Lindermayer systems (L-systems) are parallel rewriting systems that were originally presented in 1968 as a theoretical framework to mathematically model growth of multicellular organisms. Being determined by the initial word (i.e. by a finite string of symbols over an alphabet) and by a finite set of production rules that prescribe the derivation process, this formalism is closely related to the formal grammar theory developed by Chomsky, the difference though is reflected in the biological motivation of L-systems. In L-system, an application of production rules proceeds in discrete time instants in a parallel manner for each symbol, resulting in absence of a terminal alphabet in the sense of formal grammar and thus bearing a resemblance to cellular automata. This built-in parallelism not only has an essential impact on its formal properties, but also makes the study of L-systems more attractive from mathematical point of view, as change is global over the entire word.

Consider the simple production rule $a \to aa$ applied on the start symbol a that produces exponentially growing sequences in each step. Acquired language is clearly $\{a^{2^i} \mid i \geq 0\}$, which is not obtainable by a sequential context-free grammar. Of course, inclusion relations between parallel languages and main language classes of the Chomsky hierarchy were profoundly studied. Some results on context-free L-languages can be found e.g. in [2].

During an intensive initial study period, L-systems were also categorised by the type of complexity of grammar rules they use, yielding a rich hierarchy of language classes. In this work, we consider only the simplest L-system termed *D0L*, signifying deterministic context-free (without interactions) parallel systems, which roughly means that there is exactly one production rule for each symbol which refers to that symbol only disregarding its neighbours, as applied in the previous example. D0L-systems, being one of the fundamental L-families, provide wide ground for research of many interesting mathematical and computer science

subjects, many of which do not occur in the "classical" formal grammar theory. We shall deal with the notion of bispecial factor, related to study of a combinatorial structure of a language, more specifically, used as a tool e.g. for calculating the factor complexity of a language (i.e. a function $C(n)$ that equals the number of distinct factors of length n for a given sequence) and the critical exponent (i.e. the supremal number of times any consecutive subsequence is repeated in an infinite word). The proper terminology will be developed later in the text.

The aim of this work is to design an algorithm enumerating all bispecial factors in circular non-pushy D0L-systems presented in [1] and then implement it in SAGE, an open-source computer algebra system [3]. As it has a rather extensive theoretical background, we will build mathematical apparatus gradually in the following chapters.

This work is organized as follows: In Chapter 1 we introduce all necessary notations and definitions. Chapter 2 provides a brief overview of some basic properties and known algorithms to determine them, namely we shall introduce the notion of finiteness of D0L language and the notions of pushy and circular D0L-system. Chapter 3 summarizes needed results on the algorithm that is to be analysed and implemented. In Chapter 4 we analyse the results from Chapter 3 and design an algorithm that is the aim of this thesis. Then, in Chapter 5 and 6, we deal with the implementation and testing of our algorithm in SAGE.

Chapter 1

Preliminaries

In this chapter we introduce the necessary notational matters and basic terminology used in the following text. The first section is devoted to the standard notation regarding D0L-systems. The second section deals with basic terms from graph theory, especially with directed graphs.Formalisms are taken from [4],[5] and [6].

1.1 Combinatorics on words

An *alphabet* Σ is a finite set of symbols or letters with cardinality denoted as $\#\Sigma$. A *finite word* w of length $|w| = n$, $n \in \mathbb{N}$, over an alphabet Σ is a sequence of symbols $w = a_0a_1 \dots a_{n-1}$ from Σ. We use $alph(w)$ to denote the smallest alphabet $\Sigma' \subseteq \Sigma$ such that w is a word over Σ'. The word of length zero is the *empty word* ϵ. The concatenation of the word w and $v = b_0b_1 \dots b_{m-1}$ is the word $wv = a_0a_1 \dots a_{n-1}b_0b_1 \dots b_{m-1}$.

The set of all finite words (resp. non-empty finite words) is denoted Σ^* (resp. Σ^+). If $\mathbf{u} = u_0u_1 \dots$ is an infinite sequence of $u_i \in \Sigma$, we call it an *infinite word* over Σ and Σ^ω is the set of all infinite words over Σ.

If $w = v_1uv_2$ for some $w, v_1, u, v_2 \in \Sigma^*$, then v_1 is a *prefix*, u is a *factor*, and v_2 is a *suffix* of w. In this case we put $(v)^{-1}u = w$ and $u(w)^{-1} = v$. By $sub(w)$ we denote the set of all factors (subwords) of w. Analogously for infinite words.

An infinite word $\mathbf{u}$ is *eventually periodic* if there exist words $v, w \in \Sigma^*$ such that $\mathbf{u} = vwww \dots = vw^\omega$; if $v = \epsilon$, $\mathbf{u}$ is *(purely) periodic*. Any infinite word that is not eventually periodic is *aperiodic*.

A *morphism* is a map $\varphi : \Sigma^* \to \Sigma^*$ that fulfills $\varphi(wv) = \varphi(w)\varphi(v) \in \Sigma^*$ for all $w, v \in \Sigma^*$. If moreover $\varphi(a) \neq \epsilon$ for all $a \in \Sigma$, the morphism φ is called *non-erasing*.

Given a morphism φ on Σ^*, if $\varphi(a)$ is not a suffix of $\varphi(b)$ for any distinct $a, b \in \Sigma$, then φ is said to be *suffix-free*. *Prefix-free* morphisms are defined analogously. If a morphism is both prefix and suffix-free, then it is *bifix-free*.

A morphism φ is prolongable on a letter a if a is a prefix of $\varphi(a)$.

If $\varphi(\mathbf{u}) = \mathbf{u}$ for some $\mathbf{u} \in \Sigma^\omega$, then $\mathbf{u}$ is a *fixed point* of φ.

If a morphism φ is non-erasing on Σ such that for some $a \in \Sigma$ and $w \in \Sigma^+$ we have $\varphi(a) = aw$ then clearly, for all n we get

$$\varphi^n(a) = aw\varphi(w)\varphi^2(w)\dots\varphi^{n-1}(w)$$

and, consequently, the infinite word

$$\varphi^\omega(a) = aw\varphi(w)\varphi^2(w)\varphi^3(w)\dots$$

is a fixed point of φ. A morphism φ on Σ is *primitive* if there exists $k \in \mathbb{N}$ such that for any pair of (possibly equal) letters $a, b \in \Sigma$ the word $\varphi^k(a)$ contains b as its factor.

Definition 1 D0L-system *is a triple $G = (\Sigma, \varphi, \omega)$, where Σ is an alphabet, φ a morphism on Σ, and $w \in \Sigma^+$ is an initial string referred to as the* axiom.

The language *of a D0L-system G denoted by $\mathcal{L}(G)$ is the set of all words generated from the axiom, i.e. $\mathcal{L}(G) = \{\varphi(\omega)^i \mid i \geq 0\}$.*

D0L-system is finite *if $\mathcal{L}(G)$ is a finite set, otherwise it is* infinite.

If φ is non-erasing, then the system is called propagating*, or shortly* PD0L.

1.2 Introduction to graph theory

A *directed graph (digraph)* of *order* n is a pair $DG = (V, E)$ where $V = \{v_1, \dots, v_n\}$ is a finite set of vertices (or nodes) and $E = \{e_{ij} = \langle v_i, v_j \rangle\} \subseteq V \times V$ is an ordered pair of vertices called a directed edge, where v_i is the initial and v_j is the terminal vertex. An edge $\langle v_i, v_j \rangle$ is said to be *incident from* v_i and *incident to* v_j. A directed edge $\langle v_i, v_i \rangle$ is a *loop*. A directed *multigraph* is a digraph with parallel edges, i.e. E is a multiset. A graph is called *simple* if it does not contain loops and parallel edges.

The *in-degree* of a vertex v, denoted $deg^-(v)$, is the number of edges with v as their terminal vertex, the *out-degree* of a vertex v, denoted $deg^+(v)$, is the number of edges with v as their initial vertex.

A directed *walk* in the graph is a finite sequence $v_{i0}, e_{j1}, v_{i1}, e_{j2} \dots e_{jk}, v_{ik}$. It is allowed to visit a vertex and an edge more than once. A directed walk is a directed *trail* if edges do not repeat. A directed trail is a directed *path* if any vertices do not repeat except for a possibly closed directed path where the initial and terminal vertices are the same. A directed closed path is a directed *cycle*. Digraph is *acyclic* if it does not contain directed cycles.

A graph $DG^* = (V^*, E^*)$ is a *subgraph* of DG if $V^* \subseteq V$ and every edge of DG^* is also edge of DG. The *transpose* or *reverse* of a directed graph

DG is another directed graph $DG^R = (V, E^R)$ where if DG contains the edge $\langle v_i, v_j \rangle$, then DG^R contains the edge $\langle v_j, v_i \rangle$. We say that a digraph is *strongly connected* if there exists a directed path from v_i to v_j for every pair $v_i, v_j \in V$.

A maximal strongly connected subgraph, i.e. a subdigraph induced on a maximal set of mutually reachable vertices, is a *strong component* of digraph. If each strongly connected component C_i of DG is contracted to a single vertex, the resulting graph $DG^C = (V^C, E^C)$ where $V^C = \{C_1 \dots C_m\}$ and $e_{k,l} \in E^C \Leftrightarrow \exists v \in C_k, w \in C_l, \langle v, w \rangle \in E$; is a *condensed digraph* (condensation) of DG.

Chapter 2

Non-pushy and circular D0L-systems

This chapter is devoted to prerequisites for functionality of the main algorithm generating bispecial factors in a *non-pushy* and *circular* D0L-system. Both notions are presented in the following subsections. Because there is no known algorithm that would decide if an arbitrary D0L-system is circular or not in finite steps, we will closely follow especially the work of Ehrenfeucht and Rozenberg [5] containing the proof of decidability of *repetitiveness* of a language of a D0L-system language. We shall use this result, as it was shown in [1] that circularity is equivalent to non-repetitiveness for non-pushy D0L-system with an injective morphism.

Definition 2 *Let w be a nonempty finite word. Any finite prefix v of $w^\omega = www...$ is a* power *of w. We denote this by $v = w^r$, where $r = \frac{|v|}{|w|}$.*

Definition 3 *Let G be a D0L-system. If for each positive integer n there is a word w such that $w^n \in sub(\mathcal{L}(G))$, then G is* repetitive*; if there is a word w such that $w^n \in sub(\mathcal{L}(G))$ for all positive integers n , then G is* strongly repetitive*.*

It is shown in [5] that all repetitive D0L-systems are strongly repetitive. The converse is obvious.

2.1 Finiteness of language of a D0L-system

It was proved already in early 1970s that finiteness of a D0L-system is decidable. For our implementation we use a criterion proposed by Vitányi in [7], but before stating it, let us first mention a classification of the letters in an alphabet Σ with respect to morphism φ as follows:

Definition 4 *Let $G = (\Sigma, \varphi, \omega)$ be a D0L-system and $a \in \Sigma$.*

(i) *We say that a is* mortal *if $\varphi^i(a) = \epsilon$ for some $i \geq 1$; $\mathcal{M}$ is the set of all mortal letters;*

(ii) alive *if $a \in \Sigma \setminus \mathcal{M}$ for some $i \geq 1$; $\mathcal{V}$ is the set of all alive letters;*

(iii) recursive *if $\varphi^i(a) \in \Sigma^* a\ \Sigma^*$ for some $i \geq 1$; $\mathcal{R}$ is the set of all recursive letters;*

(iv) monorecursive *if $\varphi^i(a) \in \mathcal{M}^* a\ \mathcal{M}^*$ for some $i \geq 1$; $\mathcal{MR}$ is the set of all monorecursive letters;*

(v) accessible *from a word $w \in \Sigma^+$ if $\varphi^i(w) \in \Sigma^* a\ \Sigma^*$ for some $i \geq 1$; $\mathcal{A}(w)$ is the set of all letters accessible from w.*

To explain the above definition more intuitively, consider the D0L-system $G = (\Sigma, \varphi_A, \omega)$, where

$$\varphi_A = \{a \to b, b \to dc, c \to ebe, d \to de, e \to fd, f \to g, g \to gh, h \to i, i \to \epsilon\}$$

and its corresponding dependence graph in Figure 2.1(a). The *dependence graph* of φ_A, denoted by $DG(\varphi_A)$ is a directed multigraph of order $\#\Sigma$ with loops allowed, its vertices are elements of $\#\Sigma$ and there exists an edge incident from a to b only if $a \in alph(\varphi(b))$.

It is easy to see that the set of letters accessible from w can be constructed recursively as

$$\mathcal{A}_1(w) = \{a_1 \mid a_1 \in alph(\varphi(w))\}$$
$$\mathcal{A}_j(w) = \mathcal{A}_{j-1} \cup \{a_j \mid a_j \in alph(\varphi(b)), b \in \mathcal{A}_{j-1})\}$$

and that there exists $k \leq \#\Sigma$ such that $\mathcal{A}(w) = \mathcal{A}_k(w)$. It suffices to notice that $\forall i > k : \mathcal{A}_i(w) = \mathcal{A}(w) \subseteq \Sigma$ as

$$k \leq \max\{\textit{the length of shortest path from } a \textit{ to } b \mid a \in alph(w), b \in \Sigma\},$$

which gives us a halting condition for search of k and at the same time a trivial algorithm for construction of $\mathcal{A}(w)$. By continually applying the morphism φ on w, we can simulate a parallel graph traversal, i.e. starting from all vertices $a \in alph(w)$, until no new vertex can be visited.

Using the previous result, we can construct other sets from Definition 4. Cycles in a dependence graph clearly represent the set of recursive letters and so $\mathcal{R} = \{a \in \Sigma \mid a \in \mathcal{A}(a)\}$. A letter a is mortal if $deg^+(a) = 0$ or if for all edges incident from a to b, $a \neq b, b$ is mortal: $\mathcal{M} = \{a \in \Sigma \mid \mathcal{A}(a) \cap \mathcal{R} = \emptyset\}$.The set of alive letters is $\mathcal{V} = \{a \in \Sigma \setminus \mathcal{M}\}$. Finally, the set of monorecursive letters is any strong component C_i^{MR} of $DG(G)$ that is a simple cycle and considering a condensed digraph $DG^C(G)$ (see Figure 2.1(b)), if there exists a path from C_i^{MR} to C_j for any j, then C_j is a vertex representing a mortal letter: $\mathcal{MR} = \{a \in \mathcal{R} \mid \forall b \in \mathcal{A}(a) : \varphi(b) \in \mathcal{M}^* c \mathcal{M}^*$, where $c \in \mathcal{R} \cup \{\epsilon\}\}$.

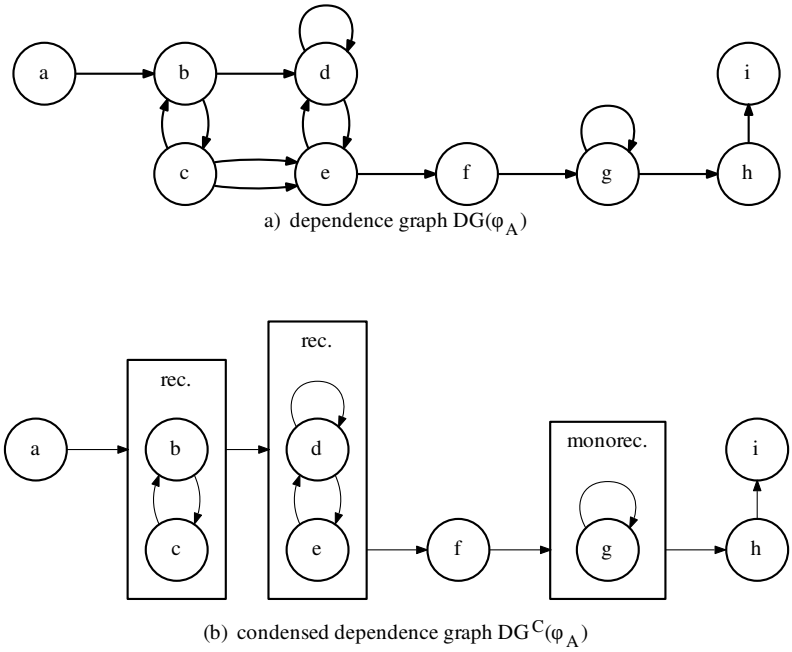


Figure 2.1: The dependence graph DG (for morphism φ_A) and its condensation

To conclude our example, in the D0L-system $G = (\Sigma, \varphi_A, \omega)$, where $\omega = cdc$, $\mathcal{V} = \{a, b, c, d, e, f, g\}$, $\mathcal{M} = \{h, i\}$, $\mathcal{R} = \{b, c, d, e, g\}$, $\mathcal{MR} = \{g\}$ and $\mathcal{A}(\omega) = \{b, c, d, e, f, g, h, i\}$.

From now on we will assume Σ to be the smallest alphabet of the language of a D0L-system $G = (\Sigma, \varphi, \omega)$ as for any D0L-system $G' = (\Lambda, \varphi, \omega)$ where $\Sigma \subset \Lambda$, $a \notin \mathcal{A}(\omega)$ for all $a \in \Lambda \backslash \Sigma$, hence G and G' generate the same language.

Lemma 1 (Vitanyi [7]) *A language of a D0L-system is finite if and only if all recursive letters which are accessible from the axiom (i.e., which occur in words in the language) are monorecursive.*

Having previously found the construction method for the sets of Definition 4, we can see that an algorithm determining if a D0L-system is finite or not is based on set operations.

2.2 Non-pushy D0L-systems

Definition 5 *Let $G = (\Sigma, \varphi, \omega)$ be a D0L-system. A letter $b \in \Sigma$ has* rank zero *in G if $\mathcal{L}(G_b)$ is finite, where $G_b = (\Sigma, \varphi, b)$. We denote the set of all*

letters having rank zero as Σ_0^*.

Definition 6 *A D0L-system* $G = (\Sigma, \varphi, \omega)$ *is pushy if* $sub(\mathcal{L}(G)) \cap \Sigma_0^*$ *is infinite; otherwise* G *is non-pushy.*

It is obvious from the above definition that a language of a pushy system must be infinite, as it must contain infinitely many arbitrary long factors over Σ_0. There are two types of letters according to Definition 4 that have rank zero - monorecursive and mortal, but both of them will hardly contribute to the growth of rank zero subsequence to infiniteness as the first will translate to itself and the latter to ϵ in at most $\#\Sigma$ morphism applications. Now, an inspection of production rules of alive letters other than monorecursive is in order and we can already expect the search for rank zero prefixes or suffixes of their production rules. This approach was formalized in [8].

Theorem 1 (Ehrenfeucht and Rozenberg [8]) *It is decidable whether or not an arbitrary D0L-system is pushy.*

In the proof of the above theorem in [8], the authors propose that a D0L-system is pushy if and only if it satisfies a so-called *edge condition*, i.e. if the following holds: there exists $a \in \Sigma$, $k \in \mathbb{N}^+, w \in \Sigma^*$ and $u \in \Sigma_0^+$ such that $alph(u)$ contains an alive letter and either $\varphi^k(a) = wau$ or $\varphi^k(a) = uaw$; and also show that the edge condition is decidable.

To detect if the edge condition is satisfied, we only need to consider two subgraphs of D0L-system dependence graph denoted as $GP(G)$ and $GS(G)$. Their vertices are elements of the set of letters having rank zero and there is an edge from a to b with label u_i if there exist u_i, w_i such that $\varphi(a) = u_i b w_i$ for $GP(G)$ or $\varphi(a) = w_i b u_i$ for $GS(G)$. Then, if there exists a cycle of length k and suffix or prefix label contributions $alph(u \mid u_{k-1}u_{k-2}\dots u_0) \cap \mathcal{MR} \neq \emptyset$, the D0L-system is pushy.

Let us consider the D0L-system $G = (\{a, b, c, d\}, a \to bacba, b \to caab, c \to de, d \to c, e \to \epsilon, ba)$ and graphs $GP(G)$ and $GS(G)$ in Figure 2.2, where prefix (for $GP(G)$) and suffix (for $GS(G)$) contributions are marked as labels of corresponding edge.

We can see that if $GP(G)$ or $GS(G)$ contains a cycle with a non-empty edge label, the D0L-system is pushy. In our case, label c of edge $\langle b, a\rangle$ in the cycle of $GP(G)$. Analogously, $G = (\{a, b, c, d\}, a \to bacba, b \to aab, c \to de, d \to c, e \to \epsilon, ba)$ is non-pushy.

2.3 Repetitiveness and circularity

The first algorithm to decide if a non-pushy system is repetitive as presented in [5] is of an unknown complexity and it consists of rather complicated construction and examination of a finite set of special D0L-systems. See the

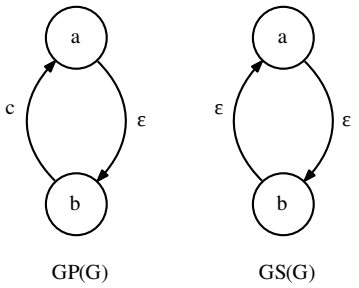


Figure 2.2: Detection of edge condition satisfiability

original paper for more information. We will use work [9], where authors prove that a non-pushy D0L-system is repetitive if and only if there is a letter a and an integer ℓ such that the fixed point of φ^ℓ starting in a is purely periodic and also provide an algorithm enumerating all infinite repetitions.

In the following subsections we shall briefly describe a simplified version of this algorithm, together with finally providing a formal definition of circular D0L-systems. Gently modified pseudocode from [9] to suit our needs can also be found below.

2.3.1 Simplifications

Definition 7 *Let Σ and Δ be two finite alphabets, and let $f : \Sigma^* \to \Sigma^*$ and $g : \Delta^* \to \Delta^*$ be morphisms. We say that f and g are* twined *with respect to (h, k), if there exist morphisms $h : \Sigma^* \to \Delta^*$ and $k : \Delta^* \to \Sigma^*$ satisfying the equalities $f = k \circ h$ and $g = h \circ k$. If $\#\Delta < \#\Sigma$ and f and g are twined, then g is called a* simplification *of f. If G does not have a simplification it is called* elementary*.*

Authors often assume that a D0L-system is elementary and thus injective and non-erasing as for any non-injective D0L-system it is possible to construct its elementary simplification and proceed with considering only elementary D0L-system if the studied property of the D0L-system in question does not change as it is in the case of e.g. repetitions; see [10]. The algorithm to create a non-erasing and injective simplification can be found in the work of Kobayashi and Otto [10].

Lemma 2 (Kobayashi and Otto [10]) *Let $f : \Sigma^* \to \Sigma^*$ be a morphism such that $\Gamma_1 := \{a \in \Sigma \mid f(a) = \epsilon\} \neq \emptyset$; and $\Delta := \Sigma \setminus \Gamma_1$, $f_0 : \Delta^* \to \Delta^*$ such that $f_0(a) := \pi_\Delta(f(a))$, $\pi_\Delta : \Sigma^* \to \Delta^*$ is the natural projection. Further, let $k : \Delta^* \to \Sigma^*$ denote the morphism defined by $k(a) := f(a)$, $a \in \Delta$.*

The morphisms f and f_0 are twined with respect to $(\pi_\Delta; k)$; and f_0 is a simplification of f.

The lemma implies that by simply removing all mortal letters and respective production rules from a D0L-system with an erasing morphism, we get a non-erasing simplification $\varphi'^m : (\Sigma \setminus \mathcal{M})^* \to (\Sigma \setminus \mathcal{M})^*$ of $\varphi^m : \Sigma^* \to \Sigma^*$, that is, we apply the last lemma m times, where $m = \min\{n \mid \forall a \in \mathcal{M}, \varphi^n(a) = \epsilon\}$. Morphisms φ^m and φ'^m are twined with respect to $(\pi_{\Sigma\setminus\mathcal{M}}, k')$, where $k' : (\Sigma \setminus \mathcal{M})^* \to \Sigma^*$ and $k'(a) = \varphi^m(a)$ for $a \in \Sigma \setminus \mathcal{M}$. The proof of that can be found in Lemma 4.1 of [10].

In the same paper the authors also present a construction of simplification of a non-erasing morphism, related to the defect theorem and notions of the coding theory.

As it will be shown later that we must restrict ourselves on injective morphisms only, let us mention a known algorithm to determine if a morphism is injective or not.

Sardinas-Patterson algorithm [11],[12]
It is known that a morphism is injective only if the set of letter images forms a code. To determine if a morphism is injective, we might use Sardinas-Patterson algorithm, a classical algorithm for determining in polynomial time whether a given variable-length code is uniquely decodable [12].

Definition 8 *Let Σ be an alphabet. A* code *over the set Σ is a subset $C \in \Sigma^+$ such that for any two sequences $x_1x_2\dots x_n, y_1y_2\dots y_m$ over C ($x_i \in C$ and $y_j \in C$; $i < n; j < m$) satisfying $x_1x_2\dots x_n = y_1y_2\dots y_m$ we have $n = m$ and $x_i = y_i$ for $\forall i \le n$.*

An element of C is called a codeword.

In other words, a set $C \in \Sigma$ is a code if any word over C^+ can be written uniquely as a concatenation of words from C, i.e. a code has the property of an unique decipherability, thus it never contains the empty word.

Before providing the pseudocode we must define the family of sets C_n referred to as sets of *dangling suffixes*, i.e. if w_i is a prefix of w_j the dangling suffix of w_j is a word $w_i^{-1}w_j$.

$$C_0 = \{w \in \Sigma^+ \mid c \in C : cw \in C\}$$

$$C_{i+1} = \{w \in \Sigma^+ \mid c \in C : cw \in C_i\} \cup \{w \in \Sigma^+ \mid c \in C_i : cw \in C\}$$

The idea of the Sardinas-Patterson algorithm is to try to "construct a word" that proves that the images of letters are not a code. We say "construct a word" as algorithm itself does not keep the information about how this word was constructed. It works with sets, thus losing dependencies between C_i and C_{i+1}. The set with all the dangling suffixes of images of

letters is built in first step (C_0), then recursively other sets are constructed from testing if dangling suffix is a prefix of some codeword of has some codeword as its prefix, building towards the searched "word".

Algorithm 1 Sardinas-Patterson algorithm

Input : a morphism $\varphi : \Sigma^* \to \Sigma^*$
Output: *True* if injective; *False* otherwise

if $\varphi(a) = \epsilon$, for any $a \in \Sigma$ **then**
    return *False*
$C \leftarrow a$ image, for all $a \in \Sigma$
$i \leftarrow 1$
compute C_0
while C_i does not contain an empty word **do**
    $i \leftarrow i + 1$
    compute C_i
    if there exists $j < i : C_j = C_i$ **then**
        return *True*
return *False*

Since all sets C_i are finite and obtaining the same set twice causes the algorithm to stop, the algorithm must always terminate. The total number of suffixes processed is at most equal to the sum of the lengths of all codewords. The algorithm runs in a polynomial time as function of input length.[12]

2.3.2 Circular D0L-systems

Definition 9 *Let φ be a morphism with a fixed point $\mathbf{u}$, φ injective on $\mathcal{L}(\mathbf{u})$, and let w be a factor of $\mathbf{u}$. An ordered pair of factors (w_1, w_2) is called a* synchronizing point *of w if $w = w_1 w_2$ and*

$$\forall v_1, v_2 \in \Sigma^*, (v_1 w v_2 \in \varphi(\mathcal{L}(\boldsymbol{u})) \implies v_1 w_1 \in \varphi(\mathcal{L}(\boldsymbol{u})) \text{ and } w_2 v_2 \in \varphi(\mathcal{L}(\boldsymbol{u})))$$

We denote this by $w = w_1|_s w_2$.

The above definition is not very intuitive, so we shall demonstrate its meaning on an example.

Consider the well-known Thue-Morse substitution: $\varphi_{TM} : 0 \to 01, 1 \to 10$ and its fixed point $\mathbf{u}_{TM} = \varphi^\omega(1) = 1001011001101001011010 0 \ldots$.

Any factors containing $0|_s 0$ and $1|_s 1$ can be decomposed into φ_{TM}-images of letters, in this case incomplete φ_{TM}-images of letters 1 and 0 for $0|_s 0$ and 0 and 1 for $1|_s 1$. This decomposition may not always be unique, e.g. in some cases substitution might not be suffix or prefix-free and we might not know to which image of a letter a factor relates to. If synchronization is the same for all such cases, a synchronizing point does exist. Considering the factor 01, it

has no synchronizing point as we can decompose it to both 0|1 and |01| and those do not share the "common bar".

Definition 10 *A D0L-system $G = (A, \varphi, \omega)$ is* circular *if φ is injective on $sub(\mathcal{L}(G))$ and if there exists $D \in N$ such that any $v \in sub(\mathcal{L}(G))$ with $|v| \geq D$ has at least one synchronizing point. This D is called a synchronizing delay.*

Obviously, synchronizing delay represents an upper bound on the length of prefixes and suffixes of all words $w \in sub(\mathcal{L}(G))$ for which we cannot unambiguously determine their (possibly incomplete) φ-images.

To continue the above example, we will show that $G = (\{0,1\}, \varphi_{TM}, 1)$ is circular. We still have to consider the factor 0101 (and analogously 1010), which we can decompose as 0|10|1 and 01|01. The decomposition to 0|10|1 is not admissible as it would imply that factor 111 is in the language of $\mathcal{L}(\mathbf{u}_{TM})$. Clearly, that is not true. As a result, the synchronizing point of 0101 is $01|_s01$, the φ_{TM}-image of 00. With that we have iterated and found synchronizing points for all admissible factors of length 4, as other cases would contain factors 00 and 11 for which the decomposition was already shown. Now we see that the searched synchronizing delay equals 4.

As we have previously mentioned, there is no known algorithm to compute the minimal or at least reasonably low synchronizing delay, and hence determine if arbitrary D0L-system is circular or not. However, we are only concerned with non-pushy D0L-systems for which circularity was proved to be decidable. A known algorithm is presented in the following subsection. One more important result on circularity was presented in work of Mossé and it states:

Theorem 2 *(Mossé [13]) Any D0L-system $G = (\Sigma, \varphi, a)$ with φ primitive and $\varphi^\omega(a)$ being aperiodic is circular.*

We should also mention that though it is easy to determine if φ is injective, to determine if φ is injective on $sub(\mathcal{L}(G))$ we must also determine if for any w_1, w_2 such that $\varphi(w_1) = \varphi(w_2)$ both w_1 and w_2 are factors of a language of a D0L-system. Of course, if φ is injective $\implies$ φ is injective on $sub(\mathcal{L}(G))$, but the converse is not true. The injectivity on $sub(\mathcal{L}(G))$ remains an open problem. From now on, we will work with circular D0L-systems with an injective morphism.

2.3.3 Algorithm determining if non-pushy D0L-system is circular

In [9], the authors presented the algorithm enumerating all infinite repetitions in a D0L-system. The algorithm uses the notion of injective simplification of a D0L-system, as it was proved in [10] that the structure of infinite repetitions

remains the same and can be retroactively constructed for the original D0L-system. The proof of the correctness of algorithm is well described in [9].

Definition 11 *A non-empty word v is* primitive *if $v = w^k$ implies $k = 1$.*

Lemma 3 *(Klouda and Starosta [9]) Any repetitive D0L-system G contains a finite number of primitive words v such that v^ω is an infinite periodic factor, i.e. $\forall k, v^k \in sub(\mathcal{L}(G))$.*

The following theorem states that non-circular D0L-systems must contain unbounded power of some word.

Theorem 3 *(Mignosi and Séébold) If a D0L-system is k-power-free (i.e., $\mathcal{L}(G)$ does not contain the k-power of any word) for some $k \geq 1$, then it is circular.*

If a D0L-system is repetitive, the algorithm in [9] outputs a list of all primitive words v such that v^k is in the language for all $k \in \mathbb{N}$. For our purposes we only need to know if at least one such primitive word is a factor of a language of a non-pushy D0L-system, thus if it is repetitive and as such non-circular.

Algorithm 2 Circularity test for a non-pushy D0L-system $(\Sigma, \varphi, \omega)$

Input: injective morphism $\varphi : \Sigma^* \to \Sigma^*$
Output: True if circular; False otherwise

```
1: calculate the list of rank zero letters Σ_0
2: for all letters a ∈ Σ \ Σ_0 do
3:     find the least ℓ such that FirstLetter(φ^ℓ(a)) = a
4:     if ℓ exists then
5:         find the least s such that φ^{ℓ·s}(a) contains at least two occurrences
             of one unbounded letter
6:         if φ^{ℓ·s}(a) contains at least two occurrences of a then
7:             v ← the longest prefix of φ^{ℓ·s}(a) with one occurrence of a;
8:             if φ^ℓ(v) = v^m for some integer m ≥ 2 then
9:                 return False
10: return True
```

We already know from Section 2.2 how to determine Σ_0, both ℓ and s are proved to be $\leq \#\Sigma$, which gives us an upper bound for their search (lines 3 and 5) and finally, code on lines 6-8 verifies periodicity of $(\varphi^\ell)^\omega(a)$ as it was proved that $(\varphi^\ell)^\omega(a)$ is periodic if and only if $\varphi^\ell(v) = v^m$ for some integer $m \geq 2$.

2.4 Circular and non-pushy test

From the previous results we get a simple test of prerequisites for the main algorithm of this thesis that was proved to be correct and finite for non-pushy and circular D0L-systems only. Still, we must expect morphism to be injective as we do not know how to test injectivity on a language of a D0L-systems.

The logics of the test is depicted in the following diagram:

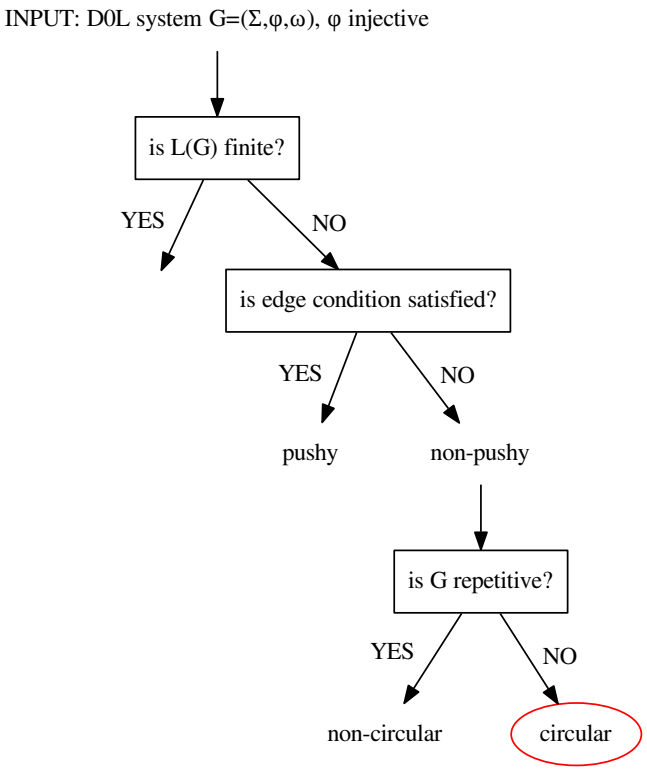


Figure 2.3: The input test for the main algorithm

Chapter 3

Bispecial factors in circular non-pushy D0L languages

The method presented in [1] provides a guideline for generating all bispecial factors of an infinite fixed point of a morphism. A morphism, in general, might have more fixed points with the different language, so in order to distinguish them, we treat a morphism and its particular fixed point as a D0L-system. The aim of what follows is just to summarize the needed results of [1]. All formalisms in this chapter are taken from [1] and [4].

Before we get to the method itself, let us provide a proper definition of a bispecial factor.

Definition 12 *Let w be a factor of an infinite fixed point $\mathbf{u}$ over Σ, the set of* left extensions *of w is defined as*

$$Lext(w) = \{a \in \Sigma \mid aw \in \mathcal{L}(\mathbf{u})\}.$$

If $\#Lext(w) \geq 2$, then w is said to be a left special *factor of $\mathbf{u}$.*

In an analogous way, we define the set of right extensions *$Rext(w)$ and a* right special *factor. If w is both left and right special, then it is called* bispecial.

A few more definitions are required for an exact description of the method. Before we proceed, let us provide its brief overview.

3.1 Brief description

The method allows us to describe all bispecial factors in an infinite fixed point of a morphism and it is finite by its nature. The idea behind it is to find a special mapping, refered to as *$f_{\mathcal{B}}$-image*, that allows us to "transform" any bispecial factor to another one and to find a certain set of bispecial factors

from which, with a mapping found, we can generate a sequences of bispecial factors that cover all bispecial factors in the language of a morphism. We refer to this set as *an initial set*.

1. The method is proved to work and to be finite for non-pushy and circular D0L-systems only. How to determine if a D0L-system has such properties was already presented in the previous chapter.

2. The $f_{\mathcal{B}}$-image is specified by two *graphs of left and right prolongations* (see Definition 19), built on top of left and right forky sets (see Definition 17) defined to posses the properties allowing $f_{\mathcal{B}}$-image to be applied repetitively.

3. An initial set is finite as a direct consequence of circularity. The proof that by repetitively applying f-image on its elements we can generate all bispecial factors in the language of a given fixed point, i.e in $sub(\mathcal{L}(G))$, can be found in [1].

3.2 The $f_{\mathcal{B}}$-image

3.2.1 Forky sets

The $f_{\mathcal{B}}$-image is defined not only on a bispecial factor, but it also reflects the changes of its left and right extensions. The reason for that is if we consider a right special factor and its image, the image does not have to be again right special. This happens in a situation, when the images of extensions start with the same letter. To remedy that, the longest common prefix is appended to the end of the image of the factor, but again, if a morphism is not prefix-free, the information about an extension is lost and we cannot determine if the acquired factor is right special. To keep the information about extensions after applying a morphism, we have to consider extensions longer than one letter. The following two definitions just define a proper structure to work with.

Definition 13 *Let* $\mathbf{u}$ *be an infinite word and* w *its factor. The sets of* left prolongations *of* w *is the set*

$$Lpro(w) = \{v \in \Sigma^+ \mid vw \in \mathcal{L}(\mathbf{u})\}.$$

We define the set of right prolongations $Rpro(w)$ *analogously.*

Definition 14 *Let* φ *be a morphism over* Σ *with a fixed point* $\mathbf{u}$*. Let us denote the set of unordered pairs of distinct letters as*

$$B_1 = \{(a, b) \mid a, b \in \mathcal{L}(\mathbf{u}), a \neq b\}.$$

A bispecial triplet $\mathcal{T}$ *in* $\mathbf{u}$ *is a triplet* $\mathcal{T} = ((w_1, w_2), v, (w_3, w_4))$, *where* $w_1, w_2 \in Lpro(v), w_3, w_4 \in Rpro(v)$ *and the last letters of* $(w_1, w_2) \in B_1$ *and the first letters of* $(w_3, w_4) \in B_1$.

In case of bispecial triplet we assume that either w_1vw_3 and w_2vw_4 or w_1vw_4 and w_2vw_3 are factors in the language $\mathcal{L}(\mathbf{u})$. The sets $Lpro(w)$ and $Rpro(w)$ are generally infinite.

Definition 15 *Let* φ *be a morphism over* Σ *and let* (v_1, v_2) *be an unordered pair of words from* Σ^+. *We define*

$$f_L(v_1, v_2) = \text{ the longest common suffix of } \varphi(v_1) \text{ and } \varphi(v_2),$$
$$f_R(v_1, v_2) = \text{ the longest common prefix of } \varphi(v_1) \text{ and } \varphi(v_2).$$

The idea behind forky sets is to find a proper subset of pairs of left ($\mathcal{B}_L$) and pairs of right ($\mathcal{B}_R$) prolongations, such that having a bispecial triplet $\mathcal{T} = ((w_1, w_2), v, (w_3, w_4))$ where $(w_1, w_2) \in \mathcal{B}_L$ and $(w_3, w_4) \in \mathcal{B}_R$ we can find another $(w_1', w_2') \in \mathcal{B}_L$, $(w_3', w_4') \in \mathcal{B}_R$ and bispecial triplet $\mathcal{T}' = ((w_1', w_2'), v', (w_3', w_4'))$ where

$$v' = f_L(w_1, w_2)\varphi(v)f_R(w_3, w_4).$$

As we have previously mentioned, the forky sets are the basic blocks of the $f_{\mathcal{B}}$-image mapping that we are trying to obtain. For the mapping to be properly defined and recursively applicable it must be ensured that any two left or right prolongations of any bispecial factor are included in $\mathcal{B}_L$ or $\mathcal{B}_R$ respectively. To help us define a good choice of $\mathcal{B}_L$ and $\mathcal{B}_R$ pairs, the following definition is in order.

Definition 16 *Let* (w_1, w_2) *and* (v_1, v_2) *be unordered pairs of words. We say that*

(i) (w_1, w_2) *is a* prefix (suffix) *of* (v_1, v_2) *if either* w_1 *is a prefix (suffix) of* v_1 *and* w_2 *of* v_2, *or* w_1 *is a prefix (suffix) of* v_2 *and* w_2 *of* v_1;

(ii) (w_1, w_2) *and* (v_1, v_2) *are* L-aligned *if*

$$(v_1 = uw_1 \text{ or } w_1 = uv_1) \text{ and } (v_2 = u'w_2 \text{ or } w_2 = u'v_2)$$

or

$$(v_1 = uw_2 \text{ or } w_2 = uv_1) \text{ and } (v_2 = u'w_1 \text{ or } w_1 = u'v_2)$$

for some words u, u'.

Analogously for R-aligned *pairs.*

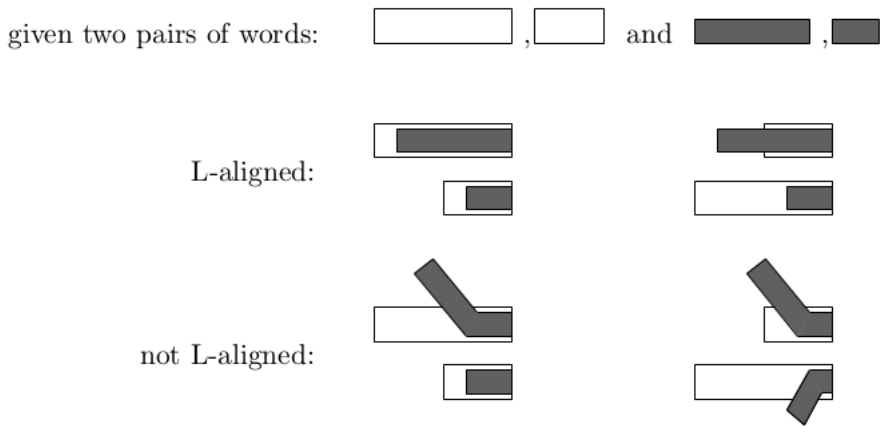


Figure 3.1: L-aligned and not L-aligned pairs of words [1]

The schematics of L-aligned and not L-aligned words is depicted in Figure 3.1 taken from [1].

With the previous definition stated, we can finally provide the definition of forky sets.

Definition 17 *(Forky sets) Let φ be a morphism with a fixed point $\mathbf{u}$. A finite set $\mathcal{B}_L$ of unordered pairs (w_1, w_2) of nonempty factors of $\mathbf{u}$ is called* L-forky *if all the following conditions are satisfied:*

(i) the last letters of w_1 and w_2 are different for all $(w_1, w_2) \in \mathcal{B}_L$,

(ii) no distinct pairs (w_1, w_2) and (w_1', w_2') from $\mathcal{B}_L$ are L-aligned,

(iii) for any $v_1, v_2 \in \mathcal{L}(\boldsymbol{u}) \setminus \{\epsilon\}$ with distinct last letters there exists $(w_1, w_2) \in \mathcal{B}_L$ such that (w_1, w_2) and (v_1, v_2) are L-aligned,

(iv) for any $(w_1, w_2) \in \mathcal{B}_L$ there exists $(w_1', w_2') \in \mathcal{B}_L$ such that

$$(w_1 f_L(w_1, w_2), w_2 f_L(w_1, w_2))$$

is a suffix of $(\varphi(w_1), \varphi(w_2))$.

Analogously we define an R-forky *set.*

The conditions $(i) - (iv)$ will be our "checkpoints" in the construction of the forky sets in the next chapter.

Definition 18 *Denote $\mathcal{B} = (\mathcal{B}_L, \mathcal{B}_R)$. The bispecial triplet $\mathcal{T}'$ in the previous text is called the $f_{\mathcal{B}}$-*image *of a bispecial triplet $\mathcal{T} = ((w_1, w_2), v, (w_3, w_4))$.*

Remark: In the following chapter we will also use the notation of f-image instead of $f_{\mathcal{B}}$-image. By that we simply mean that corresponding sets of pairs from $\mathcal{B}_X$ might not map properly on $\mathcal{B}_X$, which is why we omit the subscript. Apart from that, the idea of mapping is the same and applicable not only to triplets. The f-image of e.g. $(v, (w_3, w_4))$ is intuitively $(vf_R(w_3, w_4), (f_R(w_3, w_4)^{-1}\varphi(w_3), f_R(w_3, w_4)^{-1}\varphi(w_4)))$.

Theorem 4 *Let φ be a morphism on $\mathcal{L}$ with a fixed point $\mathbf{u} = \varphi^\omega(a)$. If (Σ, φ, a) is a circular non-pushy system, then it has finite L-forky and R-forky sets.*

3.2.2 Graphs of prolongations

The forky sets are proved to be well-defined and the condition (iv) specifies a unique mapping form (w_1, w_2) to (w_1', w_2') for all pairs in the set, so it can be applied repetitively as required. The uniqueness is a direct result of properties (ii) and (iii). The "transition" form (w_1, w_2) to (w_1', w_2') can be represented by two directed graphs.

Definition 19 *Let φ be a morphism with a fixed point $\mathbf{u}$ and let $\mathcal{B}_L$ be its L-forky set. We define the directed labeled* graph of left prolongations $GL_\varphi^{\mathcal{B}_L}$ *as follows:*

(i) the set of vertices is $\mathcal{B}_L$,

(ii) there is an edge from (w_1, w_2) to (w_3, w_4) if $(w_3 f_L(w_1, w_2), w_4 f_L(w_1, w_2))$ is a suffix of $(\varphi(w_1), \varphi(w_2))$. The label of this edge is $f_L(w_1, w_2)$.

We define graph of right prolongations $GR_\varphi^{\mathcal{B}_R}$ *analogously.*

To generate a new bispecial triplet from a bispecial triplet $((w_1, w_2), v, (w_3, w_4))$, we can simply append the label of the edge incident from (w_1, w_2) to (w_1', w_2') to the bispecial factor v from the left and the label of the edge incident from (w_3, w_4) to (w_3', w_4') from the right and "move" to new vertices (w_1', w_2') and (w_3', w_4') automaton-like.

Lemma 4 *Let φ be a morphism with a fixed point $\mathbf{u}$, let $\mathcal{B}_L$ be an L-forky and $\mathcal{B}_R$ an R-forky set and let $\mathcal{T} = ((w_1, w_2), v, (w_3, w_4))$ be a bispecial triplet of $\mathbf{u}$. Let us denote by*

(i) $g_L(w_1, w_2)$ the end of the edge of $GL_\varphi^{\mathcal{B}_L}$ starting in (w_1, w_2),

(ii) $g_R(w_3, w_4)$ the end of the edge of $GR_\varphi^{\mathcal{B}_R}$ starting in (w_3, w_4).

Then

$$\mathcal{T}' = (g_L(w_1, w_2), f_L(w_1, w_2)\varphi(v)f_R(w_3, w_4), g_R(w_3, w_4))$$

is also a bispecial triplet of $\mathbf{u}$.

Repeating above action n times, we get the bispecial triplet

$$(g_L^n(w_1,w_2), f_L(\varphi^{n-1}(w_1),\varphi^{n-1}(w_2))\varphi^n(v)f_R(\varphi^{n-1}(w_3),\varphi^{n-1}(w_4)), g_R^n(w_3,w_4)).$$

where $g_L^n(w_1,w_2)$ and $g_R^n(w_3,w_4)$ are the vertices reached in n steps from (w_1,w_2) and (w_3,w_4) respectively. As said before, this mapping is unique and so every forky vertex (w_1,w_2) of the graph of prolongations has $deg^+((w_1,w_2)) = 1$.

Having constructed graphs of left and right prolongations, we have a full prescription of $f_{\mathcal{B}}$-image for a particular fixed point.

3.3 Initial bispecial factors

With the $f_{\mathcal{B}}$-image specified, the last thing to acquire is the finite initial set of bispecial triplets, the minimal set of "generators" of all the other bispecial factors in the language of a particular infinite fixed point. This notion is connected to somehow weakened term of synchronizing point of a factor in a circular morphism that is fitted for bispecial triplet.

Definition 20 *Let φ be a morphism injective on $\mathcal{L}(\mathbf{u})$, where $\mathbf{u}$ is its fixed point, let $\mathcal{B}_L$ and $\mathcal{B}_R$ be L- and R-forky sets, and $\mathcal{T} = ((w_1,w_2),v,(w_3,w_4))$ a bispecial triplet. Assume, without loss of generality that $w_1vw_3, w_2vw_4 \in \mathcal{L}(\mathbf{u})$. An ordered pair of factors (v_1,v_2) is called a* BS-synchronizing point *of $\mathcal{T}$ if $v = v_1v_2$ and*

$$\forall u_1,u_2,u_3,u_4 \in \Sigma^*, (u_1w_1vw_3u_3, u_2w_2vw_4u_4 \in \varphi(\mathcal{L}(\mathbf{u})) \implies u_1w_2v_1, u_2w_2v_1, v_2w_3u_3, v_2w_4u_4 \in \varphi(\mathcal{L}(\mathbf{u})).$$

We denote this by $v = v_1|_{bs}v_2$.

It is clear that both left and right pairs of prolongations store an extra information available to factor in the triplet, so $v = v_1|_s v_2 \implies v = v_1|_{bs}v_2$ but converse is not true.

Definition 21 *Let φ be a morphism with a fixed point $\mathbf{u}$ which is injective on $\mathcal{L}(\mathbf{u})$. A bispecial triplet $\mathcal{T} = ((w_1,w_2),v,(w_3,w_4))$ is said to be* initial *if it does not have any BS-synchronizing point.*

We see the role of circularity in finiteness of algorithm, as a circular D0L-system requires from the definition that its language contains only a finite number of factors without any synchronizing point. Clearly, all initial factors have to be shorter than the synchronizing delay of a respective D0L system. The proof that a set of all initial bispecial triplets is a complete set of generators can be found in Theorem 36 of [1]. We shall state it here without the proof to conclude this section.

Theorem 5 *Let (Σ, φ, a) be a circular non-pushy D0L-system, $\mathcal{B}_L$ and $\mathcal{B}_R$ its L-forky and R-forky set, and $u = \varphi^\omega(a)$ infinite. Then there exists a finite set $\mathcal{I}$ of bispecial triplets such that for any bispecial factor v there exist a bispecial triplet $\mathcal{T} \in \mathcal{I}$ and $n \in \mathbb{N}$ such that $((w_1, w_2), v, (w_3, w_4)) = (f_B)^n(\mathcal{T})$ for some $(w_1, w_2) \in \mathcal{B}_L$ and $(w_3, w_4) \in \mathcal{B}_R$.*

Chapter 4

Analysis and Design

This chapter is devoted to the analysis of the needed results from [1] and the design of algorithms for the $f_{\mathcal{B}}$-image mapping and generator of the set of initial triplets. Consequently we design an algorithm to generate bispecial factors in the language of a fixed point of a morphism.

4.1 Forky sets

It is shown in [1] that the construction of forky sets is simple for bifix-free morphisms. In such case, $\mathcal{B}_L$ and $\mathcal{B}_R$ contain just an unordered pairs of letters of the alphabet i.e. for an n-letter alphabet we get set with cardinality $\binom{n}{2}$. We will call this set *the set of initial pairs*. Clearly, conditions $(i) - (iv)$ of Definition 17 are satisfied trivially, as combinations of letters are distinct, none of them are L- or R-aligned and they create all possible first or last letter extensions for any pair of factors in the language for R-forky or L-forky set respectively. Furthermore, $(\varphi(w_1)f_L(w_1,w_2)^{-1}, \varphi(w_2)f_L(w_1,w_2)^{-1})$ is always a pair of non-empty words and hence it always has a suffix in $\mathcal{B}_L$. The same goes for $(f_R(w_1,w_2)^{-1}\varphi(w_1), f_R(w_1,w_2)^{-1}\varphi(w_2))$ and a prefix in $\mathcal{B}_R$.

However, for morphisms that are not bifix-free, the choice of pairs for the forky sets is not as evident. The proof of Theorem 4, to be found in [1], shows the construction in a direct connection to the circularity of the associated D0L-system, as it uses the synchronizing delay to find a forky set. We will give a method to construct the minimal forky set which may be distinct from the one in the proof. This method starts from the set of initial pairs and then it modifies it recursively so that it satisfies the conditions $(i) - (iv)$.

4.1.1 Construction by example

We shall demonstrate the construction on the following example of non-pushy and circular D0L-system with morphism:

$$(\{0,1,2\}, \varphi_M : 0 \to 0120, 1 \to 012, 2 \to 01, 0).$$

Starting from the set of initial pairs $\mathcal{B}_{R_{init}} = \{(0,1),(0,2),(1,2)\}$ we see that none of its elements fulfills the condition (iv) as

$$f_R(0,1)^{-1}(\varphi_M(0),\varphi_M(1)) = (0,\epsilon)$$
$$f_R(0,2)^{-1}(\varphi_M(0),\varphi_M(2)) = (20,\epsilon)$$
$$f_R(1,2)^{-1}(\varphi_M(1),\varphi_M(2)) = (2,\epsilon).$$

Prolongations cannot be determined on the spot, we have to build up to the correct set gradually. In this case, the empty word indicates that we did not succeed to find proper right extensions of right special factors $\varphi_M(v)012$ and $\varphi_M(v)01$ and we have to extend both 1 and 2.

To find the relevant right extensions we need to determine the set of all factors of length 2. The set can be determined by a simple algorithm, presented later in this chapter, and in this case it equals $\{00,01,10,12,20\}$, hence $Rext(1) = \{0,2\}$ and $Rext(2) = \{0\}$. By replacing $(0,1)$ with $(0,10)$ and $(0,12)$, and $(0,2)$ with $(0,20)$ we get the new set to test.

$$\mathcal{B}_{R_1} = \{(0,10),(0,12),(0,20),(1,20)\}.$$

Repeating the above process, we see that we have to extend 0 in the first two elements of $\mathcal{B}_{R_1}$. 0 is extendible to 00 and 01, yielding the new quadruple of the pairs and the set

$$\mathcal{B}_{R_2} = \{(00,10),(01,10),(00,12),(01,12),(0,20),(1,20)\}.$$

Though the empty word does not have a prefix in any $\mathcal{B}_{R_i}$ from the definition, for other outcomes of $f_R(w_1,w_2)^{-1}(\varphi_M(w_1),\varphi_M(w_2))$ we have to iterate over the set and determine the correct index of an element in a pair to extend, if the pair does not have a prefix in $\mathcal{B}_{R_i}$. From

$$f_R(00,12)^{-1}(\varphi_M(00),\varphi_M(12)) = (0120,1)$$
$$f_R(01,12)^{-1}(\varphi_M(01),\varphi_M(12)) = (012,1)$$

we see that we have to extend 12 again. As the set of all factors of length 3 equals $\{001,010,012,101,120,200,201\}$, the only option is 120, which brings us to the final, R-forky set as it satisfies all defined conditions.

$$\mathcal{B}_R = \{(00,10),(01,10),(0,20),(00,120),(01,120),(10,20),(12,20)\}.$$

It follows from the construction that satisfiability of conditions $(i)-(iii)$ is never disturbed as we extend to the right and include all possible letters for that new factor is in language of the D0L-system.

Since φ_M is suffix-free, the L-forky set is

$$\mathcal{B}_L = \{(0,1),(0,2),(1,2)\}.$$

4.1.2 Algorithm

Using the example from the previous section, we can design an algorithm that constructs L- and R-forky sets by simply copying the advancement of the calculation. Clearly, we will need a function to determine the f-image of a pair, a function to determine index of an element in pair to extend and of course a function to determine all possible extensions of a word. In all three cases, we have to differentiate between L-forky suffix and R-forky prefix "search".

We shall omit naming the above functions in pseudocode for its better readability, providing just a word description, but we comment on them below the code to specify their properties and the functionality. The functions can be easily matched to the pseudocode.

Algorithm 3 Construction of R-forky set

Input : circular non-pushy D0L-system (Σ, φ, a) with a fixed point **u**
Output: R-forky set

```
1: B ← the set of initial pairs (u, v)
2: do
3:     for all (u, v) in B do
4:         ((fu, fv), f_R(u, v)) ← f-image of (u, v)
5:         if (fu, fv) does not have a prefix in B then
6:             i ← the index of element in (u, v) to extend
7:         else
8:             i ← out-of-range
9:         if i equals 0 or 1 then
10:            new_pairs ← list of extensions of (u, v) in index i
11:            replace (u, v) with new_pairs in B
12: while replacement occurred
13: return B
```

L-forky set is constructed in the same manner.

Subroutines

f-image of a pair (u, v)
Input: φ, pair (u, v), type=$left/right$
This function determines and returns the f-image of respective right or left pair (u, v), i.e., a tuple $(f_R(u, v)^{-1}(\varphi(u), \varphi(v)); f_R(u, v))$ denoted as $((fu, fv), f_R(u, v))$, analogously for the left pair. It is a straightforward function from definition; it applies the morphism φ and finds the longest common prefix or suffix according to the type of the pair.

Index function
Input: pair of prolongations (fu, fv) of f-image of (u, v), the set of pairs $\mathcal{B}$, type=$left/right$
According to the condition (iv), for any $(w_1, w_2) \in \mathcal{B}_R$ there exists the uniquely given pair $(w_1', w_2') \in \mathcal{B}_R$ such that $(f_R(w_1, w_2)w_1', f_R(w_1, w_2)w_2')$ is a prefix of $(\varphi(w_1), \varphi(w_2))$.

If a pair $(fu, fv) = f_R(u, v)^{-1}(u, v)$ contains the empty word, the function returns the index of the empty word in the pair. Otherwise it iterates over the elements of $\mathcal{B}$, detecting if $\mathcal{B}$ contains a prefix of (fu, fv). As the prefix is unique, it is easy to find. If found, the function returns `out-of-range` index.

If the prefix is not in the set $\mathcal{B}$, the function returns the index of the element of the pair (u, v) to be extended. The index is determined as follows: Let (u, v) be the tested pair and $(fu, fv) = f_R(u, v)^{-1}(u, v)$. Assume w.l.o.g. that it suffices to extend v to fulfil the condition (iv) for the given pair. Then either u' is prefix of fu and fv is prefix of v', or v' is prefix of fu and fv is prefix of u' for some element $(u', v') \in \mathcal{B}$.

Analogously for L-forky set.

Extension function and the set of all factors of length n
Input: φ, pair (u, v), type=$left/right$
We have mentioned the necessity of constructing a set of all factors of an arbitrary length to determine if a factor with an appended extension is in the language of the fixed point. As we work with a morphism, not its "infinite" fixed point[1], we cannot use the obvious tools for construction of a factor set, such as suffix tree, without generating the word long enough to surely contain all factors of given length. In general, this dependence is not known. To remove the need to generate unnecessarily long words of unknown length, we propose the following to obtain the set of all factors of length n that shall suffice for our test cases.

1. Find the least l such that $n \leq \mid \varphi^l(axiom) \mid$
2. Generate the word $\varphi^l(axiom)$
3. Create the set of all factors of length n as follows:
   (i) $S_0 \leftarrow$ all factors of length n from the word in 2.
   (ii) $S_{i+1} \leftarrow S_i \cup$ {all factors of length n from $\varphi(w) \mid w \in S_i$}
   (iii) if $S_{i+1} = S_i$, finish.

In case of a D0L-system representing a fixed point of a morphism, 3ii can be replaced by $S_{i+1} \leftarrow$ {all factors of length n from $\varphi(w) \mid w \in S_i$}. As

[1]It is a common practice not to try to store infinite words in finite memory of one's beloved computer

a morphism is prolongable in an axiom, the information from the previous set will not be lost. This is not true in general.

Having constructed the set of all factors of length n, we can easily determine if any word is in language or not. The extension function then just iterates over letters of the alphabet, checking if the extended factor is in the language.

4.2 Graphs of left and right prolongations

Graphs of prolongations are fully defined from the elements of forky sets that are the vertices of graphs, having the directed edges specified by the condition (iv). The algorithm for their construction uses the already defined f**-image** and **Index** functions. Having those, we proceed according to the Definition 19.

The f**-image** of (u, v) returns the information about the label of an edge incident from (u, v). The pair that it is incident to can be determined using the **Index** function, which returns `out-of-range` if the pair is found in the set. We can use the one element set as input argument to determine if return value of `Index` function is `out-of-range` for a given pair.

Algorithm 4 Construction of graph of right prolongations

Input: circular non-pushy D0L-system (Σ, φ, a), R-forky set $\mathcal{B}_R$
Output: directed graph of right prolongations $GR_{\varphi}^{\mathcal{B}_R}$

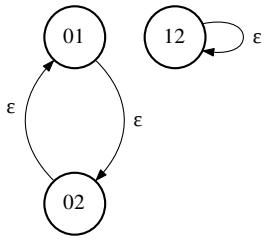
1: $GR_{\varphi}^{\mathcal{B}_R} \leftarrow \{\}$
2: $vertices \leftarrow$ all pairs (u, v) from $\mathcal{B}_R$
3: add all $vertices$ to $GR_{\varphi}^{\mathcal{B}_R}$
4: **for all** (u, v) in $vertices$ **do**
5: $\quad((fu, fv), f_R(u, v)) \leftarrow f$-image of (u, v)
6: $\quad edge\_label \leftarrow f_R(u, v)$
7: $\quad$**for all** (x, y) in $vertices$ **do**
8: $\quad\quad$**if** (x, y) is prefix of (fu, fv) **then**
9: $\quad\quad\quad GR_{\varphi}^{\mathcal{B}_R} \leftarrow$ edge from (u, v) to (x, y) with label $edge\_label$
10: $\quad\quad\quad$**continue**
11: **return** $GR_{\varphi}^{\mathcal{B}_R}$

Graph of left prolongations is constructed in the same manner.

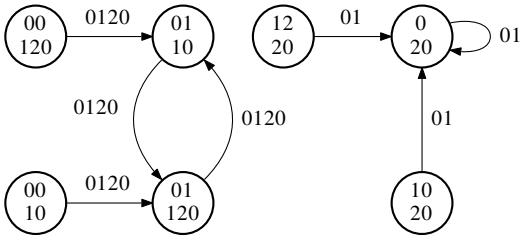
For the D0L-system from the previous example with the morphism φ_M we get the graphs in Figure 4.1.

4.3 Set of initial bispecial triplets

The notion of an initial bispecial triplet is bound to the notion of bispecial synchronizing point. We know that a length of factor of a bispecial triplet



(a) graph of left prolongations for φ_M



(b) graph of right prolongations for φ_M

Figure 4.1: $GL^{\mathcal{B}_L}_{\varphi_M}$ and $GR^{\mathcal{B}_R}_{\varphi_M}$

is less than or equal the synchronizing delay of circular morphism. Also, the triplet does not have any $f_{\mathcal{B}}$-preimage. As we do not know how to efficiently determine if bispecial factor has synchronizing point or not, we shall handle the maximum length of all bispecial factors to generate as an input variable. We will refer to it as *delay* as it is supposed to express the "safe" upper bound.

Setting the *delay* too high would imply generating extra triplets that are surely not initial. Though those can be additionally removed from the generated set, we first have to generate them by known methods which are not efficient, thus wasting time and space resources. Even knowing the exact value of *delay*, extra work has to be done as the synchronizing delay represents an upper bound on minimal needed value and we have to check for $f_{\mathcal{B}}$-preimage once the triplets are constructed. Setting the *delay* too low, our initial set would be incomplete.

A bispecial triplet might have an empty word as $f_{\mathcal{B}}$-preimage. The empty word is bispecial trivially for every alphabet with cardinality at least 2. Those are usually of no interest to us and also often "roots" of the same sequence of bispecial factors obtained from Theorem 5, so we might replace them with

their $f_{\mathcal{B}}$-images, thus removing the duplicates.

Returning to our φ_M with synchronizing delay 2, the initial set is

$$\begin{array}{lll}((0,1),\epsilon,(01,10)), & ((0,1),\epsilon,(01,120)), & ((0,1),\epsilon,(0,20)),\\ ((0,1),\epsilon,(10,20)), & ((0,1),\epsilon,(12,20)), & ((0,2),\epsilon,(00,10)),\\ ((0,2),\epsilon,(00,120)), & ((0,2),\epsilon,(01,10)), & ((0,2),\epsilon,(01,120)),\\ ((1,2),\epsilon,(0,20)), & ((0,2),0,(01,120)), & ((1,2),0,(01,120)).\end{array}$$

After replacing the empty factors with $f_{\mathcal{B}}$-image of self, we get a slightly smaller set:

$$\begin{array}{lll}((0,2),0120,(01,10)), & ((0,2),01,(0,20)), & ((1,2),0,(01,120)),\\ ((0,1),0120,(01,10)), & ((0,2),0,(01,120)), & ((0,1),0120,(01,120)),\\ ((1,2),01,(0,20)), & ((0,2),0120,(01,120)). & \end{array}$$

We might notice that the only vertices that are in the cycles of the graph $GR^{\mathcal{B}_R}_{\varphi_M}$ in Figure 4.1(b) were preserved in the modified set. That is due to the fact that we have generated a set of initial pairs from all combinations of letters in an alphabet, instead of using unordered pairs from the set of all factors of length 2. Disregarding the empty bispecial factor, the input variable *delay* would no longer have to represent synchronizing delay to get a desired set, but length of the longest factor of $f_{\mathcal{B}}$-image of bispecial triplet with the empty word as a factor. The modified graph reads:

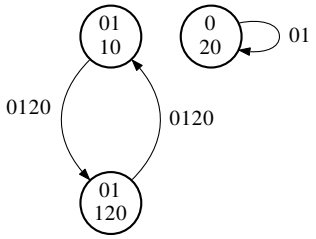


Figure 4.2: Subgraph of $GR^{\mathcal{B}_R}_{\varphi_M}$ for triplets with non-empty bispecial factors

4.3.1 Algorithm

Subroutines

Initial triplet generator

Initial bispecial factors have to be determined using any known method. We can use a simple iterator over the set of all the factors of length in range $(0, delay)$ storing the left and right extensions in format $(\{a_1 \dots a_n\}, bs, \{b_1 \dots b_m\})$, consequently creating pairs of extensions

Algorithm 5 Initial bispecial triplets

Input : D0L-system (Σ, φ, a), L- and R-forky sets, *delay*
Output: the set of initial bispecial triplets

1: generate bispecial factors with left and right extensions up to length *delay*
2: create initial triplets from the output of previous point
3: append corresponding pairs from forky sets to initial triplets to create *bispecial_triplets* ▷ pairing is uniquely given
4: **for all** *bst* in *bispecial_triplets* **do**
5: **if** *bst* factor equals ϵ **then**
6: replace *bst* with its $f_{\mathcal{B}}$-image
7: **for all** *bst* in *bispecial_triplets* **do**
8: **if** f-image of *bst* is in *bispecial_triplets* **then**
9: mark f-image of *bst* for removal
10: remove marked elements from *bispecial_triplets*
11: **return** *bispecial_triplets*

followed by inspection of their existence in $\mathcal{L}(\varphi^\omega(a))$. Such a procedure is somehow lengthy, but necessary. We can avoid generating correct extensions by simply testing all combinations of initial left and right pairs. Either way, this algorithm is not very efficient as all known methods to find bispecial factors have to iteratively search the factors to evaluate them.

4.4 Generator of bispecial factors

Every $f^n_{\mathcal{B}}$-image of an initial bispecial triplet is synchronized and has a unique preimage in the set of initial triplets. Still, the situation may occur when initial bispecial triplets might generate the same sequence of bispecial factors, resulting in duplicity as it is in case of, e.g. $((w_1, w_2), 0120, (w_3, w_4))$ from the example above. From the graphs in Figure 4.1 we see that any transition from any left pair does not contribute to left growth of factor as labels are the empty words. The factor 0120 grows to the right only and it is always extended by 0120. Generating bispecial factors from such graphs would result in generating four times the same 0120^k factor for all k. These cases should be unified to diminish a redundancy. Using our example, it suffices to keep three initial factors as from $((0, 2), 0120, (01, 10))$, $((0, 1), 0120, (01, 10))$, $((0, 1), 0120, (01, 120))$ and $((0, 2), 0120, (01, 120))$ we would generate the same bispecial factors, and the same goes for bispecial triplets $((0, 2), 01, (0, 20))$ and $((1, 2), 01, (0, 20))$ and bispecial triplets $((1, 2), 0, (01, 120))$ and $((0, 2), 0, (01, 120))$.

For this purpose, we need to examine cycles in graphs of prolongations for all bispecial triplets with the same bispecial factor and remove any unnecessary transitions and corresponding initial triplets. Then we can proceed generating

bispecial factors by "transiting" to neighbouring vertices of left and right graphs of prolongations, prepending and appending the corresponding labels to φ-image of the factor.

The set of all bispecial factors is clearly infinite, the generator must be bounded by a number of iterations or a maximum length of bispecial factor to obtain.

With this we have successfully analysed the method presented in [1] and identified the main and the auxiliary functions necessary for an implementation.

Chapter 5

Implementation

The implementation of our algorithms is to be found on attached CD. If the reader wishes to implement a new algorithm or improve upon what we have provided, this chapter discusses details of our implementation and also serves as the documentation.

5.1 Introduction to *Sage*

Sage, also known as Sagemath, is an open-source Computer Algebra System with support for Linux, Solaris and Mac OSX platforms available for download from `www.sagemath.org`. It was originally created to provide a viable alternative to commercial products such as Wolfram Mathematica® or MATLAB®. The acronym originally meant Software for Algebra and Geometry Experimentation, but today it is depreciated as Sage overgrew its initial intentions in quite a short time. Now, it is a robust system built on top of many existing open-source packages that unifies a wide range of mathematical and computer science fields such as algebra, calculus, number theory, cryptography, combinatorics, graph theory and many more. Being a very powerful tool, it is suitable to undertake computations for mathematical research and development.

Sage is a project of William Stein from University of Washington in Seattle, Washington, USA that is being developed since 2004-2005. In quite a short time Sage has been able to attract hundreds of users and developers, who have contributed to the open source base of Sage. It is partially due to the fact that in 2007, it was nominated for the French competition Trophées du Libre and won the first place in Scientific Software category, getting a lot of good publicity.

Sage has two user interfaces - interactive command line and notebook in a web browser that connects to a local Sage installation. Notebook is currently being modified to provide even better user experience and partially compensate for not having a graphical user interface of its own. On top of that,

Sage also offers a new online service for running Sage computations without need of installing the software locally - SageMathCloud, currently in its beta version.

The notebook and the cloud interfaces contain extra functions as far as graphics is concerned, e.g. graph editor or currently a very popular @*interact* functionality allowing to create interactive web applications with the community of its own.

Sage attracts new users every day as in certain areas it excels even over the commercial software as some of its algorithms are proved to be the fastest implementations for the given problems. And, it is getting better all the time.

5.2 Programming in Sage

Sage uses Python, one of the world's most popular general purpose scripting languages, as the main programming language and it not only supports all of its commands, but it also uses the syntax of Pythons. Sage, by using NumPy (the fundamental package for scientific computing with Python) for storing and manipulating numerical data and related packages (e.g. SciPy for statistics and optimization, Matplotlib for graph making, Pandas for data processing and analysis tools) gets strong scientific programming capabilities.

Besides being able to run Python scripts, Sage also features the Cython compiler. Cython provides the option of surrendering some of the dynamic features of Python in exchange for potentially huge speedups. Cython is critical to the design of Sage. It makes it possible to efficiently make use of data types and functions defined in C/C++ libraries. Basically, using Sage means using Python with a lot of extra predefined computation functionality.

5.3 Program Modules

The data types that we deal with are not too complex. We require tools available to manipulate sets of letters, arbitrarily large words, directed graphs and built upon letters and words to manage higher levels of abstraction - a morphisms. Sage, being a powerful tool already contains modules with morphisms and directed graphs (`WordMorphism` and `DiGraph` classes), even the abstract word class with representation of both finite and infinite words. Here, we will shortly discuss each of the data types as well as their associated methods that we use in our implementation of `DOLSystem` and `FixedPointBS` classes.

5.3.1 D0L class

A set of symbols is needed to represent matters such as mortal, monorecursive, etc., letters with respect to φ, the letters accessible from

$w \in \Sigma$ and rank zero letters. For set operations we use standard Python sets as Sage does not provide more beneficial data types. Sage `Alphabet` and `Set` provide an extra functionality, unfortunately only for Python's enumerated immutable frozenset type. Using Python's sets allows us to use all of its methods hence to add and remove letters, to test for difference, to test whether two sets are disjoint or convert word to a given set, determine the cardinality and so on, providing everything we need.

Our most important data structure is a morphism. Paired with an alphabet and the axiom morphism becomes a D0L-system. As previously said, morphisms have built-in support in Sage, that is since the version of 2007, but with the functionalities using more general settings than what is needed for D0L-systems. A D0L-system can be easily represented as morphism applied on an axiom, but in order to ensure consistency of the data structure, as the user is not supposed to directly modify individual attributes of an object, we isolate a D0L-system in a separate Python class, thus creating an abstract base class for Sage objects of D0L-system using the `WordMorphism` by a composition. This allows us to impose extra restrictions we demand for input data.

`DOLSystem` module contains a substantial amount of our work, representing mainly functional prerequisites for the main algorithm. As mentioned, it is a class with a "private" attributes of *alphabet*, *morphism* and *axiom*. More on input restrictions can be found in the next chapter concerned with a testing. We do not demand alphabet to be specified as it can be easily generated from a given morphism.

```
sage: DOLSystem('0->010,1->11','0')
D0L System: ({'0', '1'}, 0->010, 1->11, '0')
```

The details of implemented methods have been explained in Chapter 2. The methods are rather straightforward implementation of given pseudocodes. To familiarize reader with used naming conventions, we provide a sample of Sage calculations. The following are the functions of a morphism only, but as we did not intend to modify Sage source base we include them in a `DOLSystem` class.

```
sage: n = DOLSystem('0->1,1->1213,2->123,3->114,4->45,5->','03')
sage: n.alive_letters()
{'0', '1', '2', '3', '4'}
sage: n.mortal_letters()
{'5'}
sage: n.recursive_letters()
{'1', '2', '3', '4'}
sage: n.monorecursive_letters()
{'4'}
```

```
sage: n.accessible_from('4')
{'4', '5'}
```

For a simple input test of the main algorithm (determining if a D0L-system is pushy and circular) we have implemented *is_language_finite* test, *is_pushy* test and for our purposes only, incomplete *is_repetitive* test based on the algorithm in [9] that can be further modified to his full potential using the *nonerasing_simplification* function to be extended to *injective_simplification* function. This test is left to raise Python's standard *NotImplementedError* in case of a morphism not being injective.

A method to test D0L language for finiteness also allows us to determine rank zero letters later to be used in both *is_pushy* and *is_repetitive* methods. For *is_pushy* implementation we use Sage's `DiGraph` module that allows us to not only work with labels, but it also contains *all_simple_cycles* method used in detecting the edge condition satisfiability.

```
sage: n.is_language_finite()
False
sage: n.is_language_finite('4')
True
sage: n.is_pushy()
True
sage: n.is_repetitive()
True
sage: DOLSystem('a->aba,b->cc,c->bb','a').is_pushy()
False
sage: DOLSystem('a->aba,b->cc,c->bb','a').is_repetitive()
True
sage: DOLSystem('a->aaba,b->b','a').is_pushy()
False
sage: DOLSystem('a->aaba,b->b','a').is_repetitive()
False
```

To determine injectivity, we provide an implementation of Sardinas-Patterson algorithm, as currently there is not any injectivity test implemented in `WordMorphism` class of Sage. As we know, a D0L-system must be non-erasing for a morphism to be injective, hence we also provide *is_erasing* method that is just redefinition of *is_erasing* method of `WordMorphism` class. The same goes for *is_primitive* method.

```
sage: DOLSystem('a->abd,b->cb,c->ab,d->dcb','a').is_injective()
False
sage: DOLSystem('a->aca,b->bac,c->d,d->c','ab').is_injective()
True
```

As the test for the main algorithm is incomplete (we cannot properly test circularity) we provide it as a "private" method of D0L class, later to be used for consistency check only. With this we restrict ourselves on cases where circularity can be determined. This test is implemented using the listed methods exactly as depicted in Figure 2.3.

It was not our intention to integrate D0L class into the code base of Sage, therefore an initial version of our implementation does not contain functions "with leading and trailing single underscores" defined for Sage, especially, instead of _repr_ we use Python's standard __repr__ and __str__ and also we omit _latex_ function that can be easily defined later.

5.3.2 Fixed point of morphism class

The code dealing with bispecial factors of a fixed point of morphism is to be found in `FixedPointBS` class that is a subclass of `D0LSystem`. Clearly, fixed points are supposed to share all of the D0L-class functionality, but extra restrictions must be imposed on the input. `FixedPointBS` is not supposed to represent class of all fixed points. That would be unnecessary as we only implement bispecial factors generator for non-pushy and circular D0L-systems, representing a fixed point of morphism, with no other functionality added. We isolate the algorithm in separate Python class for reasons of presentation of the results in article [1].

```
sage: FixedPointBS('0->01,1->0','0')
D0L System: ({'0', '1'}, 0->01, 1->0, '0')
```

The main data to represent in this algorithm are unordered pairs for storing the prolongations and bispecial triplets. We represent unordered pairs as Pythons immutable but indexable type - tuple. It follows from the certain kind of symmetry which forky sets offer that the unordered pairs can be treated as ordered pairs. Once the set of initial pairs is generated in an ordered manner, i.e. the elements of pairs are lexicographically ordered, prolonging extensions preserves the lexicographic order of right pair and reverse lexicographic order for left pair. This eases the operations of equality testing as hash of (a, b) and (b, a) is of course different. The same goes for bispecial triplet and its representation by tuple of 2 tuples and a word $((w_1, w_2), v, (w_3, w_4))$. Of course, f-image of a pair is no longer ordered in general, unlike $f_{\mathcal{B}}$-image of the triplet, where pairs are obtained from the graphs. Hence, tuple representation of a bispecial triplet has all the properties we require and we do not need to define a special structure to represent it. Such a structure would not have a significant meaning beyond our program.

Again, the algorithms were described in detail in Chapter 4. Our fundamental function is a $factor\_set$ that uses Sage's $factor\_set$ function form `FiniteWord_class` on factors of images. Sage's $factor\_set$ uses the

linear-time online construction of a suffix-tree from `ImplicitSuffixTree` module to identify all factors of a given length. The method *is_in_language* uses the same principle. If we could determine a power of an image that contains all the factors of length of word on input, we could modify this function to use a suffix-tree to search for a word, thus increasing efficiency. That would be very desirable, as those are the most frequent methods we use, especially in *get_letter_extensions* and *get_word_extensions*, used both in the construction of forky sets and construction of the initial set of bispecial triplets.

As the name indicates *get_letter_extensions* and *get_word_extensions* return extensions or extended factors specified on the input. Both of them require an extra parameter to specify search for prefix or suffix.

```
sage: n=FixedPointBS('0->0120,1->012,2->01','0')
sage: n.factor_set(2)
set([word: 12, word: 20, word: 01, word: 10, word: 00])
sage: n.is_in_language('21')
False
sage: n.get_letter_extensions('01',left=False)
[word: 0, word: 2]
sage: n.get_word_extensions('01',left=False)
[word: 010, word: 012]
```

The method *graph_of_prolongations* returns graphs which together with the morphism represent the $f_{\mathcal{B}}$-image. It uses the straightforward implementation of the pseudocode 3 and its auxiliary functions specified in the previous chapter. For storing graphs we use once again Sage's `DiGraph` module. The method *get_forky_set* then simply decodes the graphs into a list of vertices.

```
sage: n.graph_of_prolongations(left=True)
Looped digraph on 3 vertices
sage: n.graph_of_prolongations(left=False)
Looped digraph on 7 vertices
sage: n.get_forky_set(left=True)
[(word: 0, word: 1), (word: 0, word: 2), (word: 1, word: 2)]
sage: n.get_forky_set(left=False)
[(word: 0, word: 20), (word: 00, word: 10),
 (word: 00, word: 120), (word: 01, word: 10),
 word: 01, word: 120), (word: 10, word: 20),
 (word: 12, word: 20)]
```

The private method *_get_triplet_fimage* uses the graphs of prolongations and the morphism to generate $f_{\mathcal{B}}$-image of the triplets. It is used in both

initial_bs_triplets and *bispecial_factors_from_fimage* which is our main generator. The *initial_bs_triplets* also requires the known generator of bispecial factors to generate the set of initials. For that we have implemented simple iterator over the set of all factors in range $(0, delay)$ that evaluates the extension function return value. This brute force method is named *bispecial_factors*.

```
sage: n.initial_bs_triplets(delay=1) # incorrect upper bound
set([((word: 0, word: 1), word: 0120, (word: 01, word: 10)),
     ((word: 1, word: 2), word: 01, (word: 0, word: 20)),
     ((word: 0, word: 2), word: 0120, (word: 01, word: 10)),
     ((word: 0, word: 2), word: 0120, (word: 01, word: 120)),
     ((word: 0, word: 2), word: 01, (word: 0, word: 20)),
     ((word: 0, word: 1), word: 0120, (word: 01, word: 120))])
sage: n.initial_bs_triplets(delay=2) # correct upper bound
set([((word: 0, word: 2), word: 0120, (word: 01, word: 10)),
     ((word: 1, word: 2), word: 01, (word: 0, word: 20)),
     ((word: 0, word: 2), word: 0, (word: 01, word: 120)),
     ((word: 0, word: 1), word: 0120, (word: 01, word: 10)),
     ((word: 1, word: 2), word: 0, (word: 01, word: 120)),
     ((word: 0, word: 1), word: 0120, (word: 01, word: 120)),
     ((word: 0, word: 2), word: 01, (word: 0, word: 20)),
     ((word: 0, word: 2), word: 0120, (word: 01, word: 120))])
```

The generator *bispecial_factors_from_fimage* requires both the maximum length n of bispecial factors to generate and *delay* to be specified on the input.

```
sage: n.bispecial_factors(n=10)
[word: , word: 0, word: 01, word: 0120, word: 01200120,
word: 012001201]
sage: n.bispecial_factors_from_fimage(n=10,delay=3)
[word: , word: 0, word: 01, word: 0120, word: 01200120,
word: 012001201]
```

The prototype of the method *bispecial_factors_from_fimage* does not include optimization discussed in the previous chapter. We use it here just to demonstrate its correctness compared to a brute force method, thus unoptimized version suffices for out test cases discussed in the next chapter.

Chapter 6

Testing

In this chapter we describe the tests that were carried out to determine a correctness of our algorithm. After providing a short description of how to run our scripts we divide the tests in two categories. The first are the user input tests and restrictions, the second are the tests based on D0L-system's properties according to the assignment.

6.1 Running the program

The program consists of the two Python scripts that can be loaded into Sage from a command-line using the load("path/to/script.py") command. Of course, `DOLSystem` superclass must be loaded first. Both `DOLSystem` and `FixedPointBS` classes then can be used as a part of Sage's installation as described in the Implementation chapter.

6.2 Exception tests

For the purposes of well-defined D0L-system, we require *morphism* to be recursively applicable. Hence, its codomain must be a subset of domain. `WordMorphism` does not impose such restriction on its input. Also, $alph(axiom)$ is checked to assure existence of corresponding rules and D0L-system is expected to be minimal. Errors in a morphism specification are handled by `WordMorphism` class.

```
sage: DOLSystem('a->ab,b->a')
------------------------------------------------------------
Traceback (most recent call last):
...
ValueError: Axiom must be specified.
sage: DOLSystem('a->ab,b->ad','a')
------------------------------------------------------------
```

```
Traceback (most recent call last):
...
ValueError: Production rule(s) missing in a->ab, b->ad
sage: DOLSystem('a->ab,b->a','c')
------------------------------------------------------------
Traceback (most recent call last):
...
ValueError: Production rule(s) for 'c' not specified.
sage: DOLSystem('a->ab,b->a,e->e','a')
------------------------------------------------------------
Traceback (most recent call last):
...
ValueError: D0L must be minimal, '{'e'}' is not accessible
from axiom.
```

As the main algorithm was proved to be correct and finite for circular and non-pushy D0L-systems that represent an infinite fixed point of morphism, we impose the following extra restrictions on the input: an infinite language, a test if D0L-system represents a fixed point, a test if D0L-system to be non-pushy and circular. Circularity is tested in a limited form as described in the previous chapters.

```
sage: FixedPointBS('a->ab,b->','a')
------------------------------------------------------------
Traceback (most recent call last):
...
ValueError: Language is not infinite.
sage: FixedPointBS('a->ab,b->bc,c->','a')
------------------------------------------------------------
Traceback (most recent call last):
...
ValueError: Must be non-pushy and circular.
sage: FixedPointBS('a->bb,b->ba','a')
------------------------------------------------------------
Traceback (most recent call last):
...
ValueError: 'a' is not a fixed point of morphism
sage: FixedPointBS('a->bb,b->ba','ab')
------------------------------------------------------------
Traceback (most recent call last):
...
TypeError: letter (=ab) is not in the domain alphabet
(={'a', 'b'})
```

6.3 Tests of properties

We have tested our functions on numerous test cases. According to Sage's specification, tests are included in the corresponding source files in example section of each public method. Also, we attach *testing.py* file with the test cases for the `FixedPointBS` class. These are a collection of examples taken from [1], [4] and elsewhere. As the outputs would be too long to list here directly, we provide just a brief overview of D0L-systems tested and the sample output of one of them.

1. Fibonacci: $G_1 = (\Sigma_2, \varphi : 0 \to 01, 1 \to 0, 0)$
2. Thue-Morse: $G_2 = (\Sigma_2, \varphi : 0 \to 01, 1 \to 10, 0)$
3. Chacon: $G_3 = (\Sigma_2, \varphi : 0 \to 0010, 1 \to 1, 0)$
4. $G_4 = (\Sigma_3, \varphi : 0 \to 0012, 1 \to 2, 2 \to 012, 0)$
5. $G_5 = (\Sigma_3, \varphi : 0 \to 0120, 1 \to 012, 2 \to 01, 0)$
6. $G_6 = (\Sigma_3, \varphi : 0 \to 012, 1 \to 112, 2 \to 102, 0)$
7. $G_7 = (\Sigma_3, \varphi : 0 \to 01, 1 \to 02, 2 \to 1, 0)$
8. $G_8 = (\Sigma_4, \varphi : 0 \to 020, 1 \to 1032, 2 \to 0201, 3 \to 032, 0)$[2]

Chacon morphism is known to be circular and not primitive. We also provide a case of a D0L-system with the morphism injective on the language, but not injective. The properties of the above D0L-systems are summarized in the following table.

D0L system	injective	primitive	synchronizing delay
1	True	True	1
2	True	True	4
3	True	False	5
4	True	True	3
5	True	True	3
6	True	True	3
7	True	True	1
8	False	True	3

As outputs of methods might be difficult to trace, we recommend comparing output of f-image generator to the brute-force $bispecial\_factors$ method in some reasonable range.

For e.g. D0L-system $(\Sigma_3, 0 \to 0012, 1 \to 2, 2 \to 012, 0)$ we get:

[2]This D0L-system will not be accepted on input. The current settings require specifying an injective morphism.

```
sage: phi_S = FixedPointBS('0->0012,1->2,2->012','0')
sage: phi_S.get_forky_set(left=False)
[(word: 0, word: 1), (word: 0, word: 2), (word: 1, word: 2)]
sage: phi_S.get_forky_set(left=True)
[(word: 0, word: 01), (word: 0, word: 012),
 (word: 0, word: 22), (word: 01, word: 2)]
sage: phi_S.graph_of_prolongations(left=False)
Looped digraph on 3 vertices
sage: phi_S.graph_of_prolongations(left=True)
Looped digraph on 4 vertices
sage: phi_S.bispecial_factors_from_fimage(n=12,delay=3)
[word: , word: 0, word: 2, word: 20, word: 012, word: 0120,
word: 20120, word: 0120012, word: 20120012, word: 012001220120]
sage: phi_S.bispecial_factors(n=12)
[word: , word: 0, word: 2, word: 20, word: 012, word: 0120,
word: 20120, word: 0120012, word: 20120012, word: 012001220120]
```

Sage also provides an alternative to our brute force method. Sage's *bispecial_factor* is a method of the class of finite words not a morphism method. This method is faster then brute force but still not suitable for generating long bispecial factors.

Conclusion

In this thesis, we have carried out a study and provide a brief overview of various properties and known results in the area of deterministic context-free L-systems (D0L-systems).

We have briefly surveyed computer algebra system Sage with respect to its support for functionalities required for D0L-systems. Based on this survey we identified relevant support features used to implement a prototype class for a D0L-system together with an algorithm generating bispecial factors in the language of a fixed point of a morphism. Sage currently provides a few features already to be used with morphisms and D0L-systems, but various algorithms are also missing.

We have provided an implementation of D0L-system class with methods testing the mentioned D0L-system properties and implementation of bispecial factors generator that after initial construction phase can generate all factors in linear time. Algorithm provides a huge speed-up compared to the known methods, but it is limited on circular and non-pushy D0L systems only.

With this implementation, we hope to lay a foundation for future work on implementing various algorithms regarding the inspection of a D0L-language structure. We hope that once the support of L-systems is fully integrated into Sage, our algorithm will be included.

Bibliography

[1] Klouda, K. Bispecial Factors in Circular Non-pushy D0L Languages. *Theor. Comput. Sci.*, volume 445, Aug. 2012: pp. 63–74, ISSN 0304-3975, doi:10.1016/j.tcs.2012.05.007. Available from: `http://dx.doi.org/10.1016/j.tcs.2012.05.007`

[2] Rozenberg, G.; Doucet, P. On 0L-Languages. *Information and Control*, volume 19, no. 4, 1971: pp. 302 – 318, ISSN 0019-9958, doi:http://dx.doi.org/10.1016/S0019-9958(71)90164-1. Available from: `http://www.sciencedirect.com/science/article/pii/S0019995871901641`

[3] Stein, W.; et al. *Sage Mathematics Software (Version 6.1.1)*. The Sage Development Team, 2014, http://www.sagemath.org.

[4] Klouda, K. *Non-standard numerations systems and combinatorics on words*. Dissertation thesis, Czech Technical University in Prague, 2010.

[5] Ehrenfeucht, A.; Rozenberg, G. Repetition of Subwords in DOL Languages. *Inf. Control*, volume 59, no. 1-3, Sept. 1984: pp. 13–35, ISSN 0019-9958, doi:10.1016/S0019-9958(83)80028-X. Available from: `http://dx.doi.org/10.1016/S0019-9958(83)80028-X`

[6] Kolář, J. *Teoretická informatika*. Česká informatická společnost, 2000, ISBN 9788090085381. Available from: `http://books.google.cz/books?id=xnFiAAAACAAJ`

[7] Vitányi, P. M. B. On the Size of D0L Languages. In *L Systems, Most of the Papers Were Presented at a Conference in Aarhus, Denmark*, London, UK, UK: Springer-Verlag, 1974, ISBN 3-540-06867-8, pp. 78–92. Available from: `http://dl.acm.org/citation.cfm?id=646593.697942`

[8] Rozenberg, G.; Salomaa, A. *Mathematical Theory of L Systems*. Orlando, FL, USA: Academic Press, Inc., 1980, ISBN 0125971400.

[9] Klouda, K.; Starosta, Š. An Algorithm Enumerating All Infinite Repetitions in a D0L System. *ArXiv e-prints*, July 2013, 1307.6408.

[10] Kobayashi, Y.; Otto, F. Repetitiveness of languages generated by morphisms. *Theoretical Computer Science*, volume 240, no. 2, 2000: pp. 337 – 378, ISSN 0304-3975, doi:http://dx.doi.org/10.1016/S0304-3975(99)00238-8. Available from: http://www.sciencedirect.com/science/article/pii/S0304397599002388

[11] Falucskai, J. Some algorithms concerning uniquely decipherable codes. *Proceedings of the 7th International Conference on Applied Informatics*, volume 2, 2007: p. 229–235.

[12] Wikipedia. Sardinas-Patterson algorithm — Wikipedia, The Free Encyclopedia. 2014, [Online; accessed 12-April-2014]. Available from: http://en.wikipedia.org/wiki/Sardinas-Patterson_algorithm

[13] Mossé., B. Reconnaissabilité des substitutions et complexité des suites automatiques. *Bull. Soc. Math. Fr.*, volume 124, no. 2, 1996: pp. 329–346.

Appendix A

Contents of enclosed DVD

- `readme.txt` ..... the file with DVD contents description
- `src` ..... the directory of source codes
  - `wbdcm` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
  - `thesis.ps` ..... the thesis text in PS format