Radboud University Nijmegen

Exploring the inference problem for D0L-Systems

Judith van Stegeren

July 8, 2013

Bachelor Thesis

Radboud University Nijmegen
Institute for Computing and Information Sciences

Supervisor
Dr. A. Silva
Radboud University Nijmegen
alexandra@cs.ru.nl

Second reader
Prof. dr. J.H. Geuvers
Radboud University Nijmegen
herman@cs.ru.nl

Contents

1 Introduction

After research spanning more than 60 years, Lindenmayer-systems, or L-systems, have evolved into elegant conceptual tools that are used in many different scientific fields. One of the core problems concerning L-systems, the so-called *syntactical inference problem*, did immediately spike the interest of researchers in various disciplines. In this thesis we will explore L-systems, and one of the subproblems of the inference problem.

Ever since Feliciangeli and Herman coined the inference problem [7], computer scientists have been trying to solve all cases of the inference problem. Solutions to this problem can give us valuable information about recurring topics in computer science research such as machine learning, information retrieval and formal language theory. However, to this day some subproblems of the inference problem are still open.

L-systems can be described as term rewriting systems or a special kind of formal grammars, consisting of a starting point (hereafter denoted as w_0) and a set of rules describing how productions can be rewritten. Instead of sequential application of rewrite rules, L-systems are parallel rewriting systems. This means that all applicable rules are being applied in one derivation step.

The syntactical inference problem poses the following question: given a sequence of derived words from an unknown L-system, it is possible to reason back to the L-system that created it?

L-systems come in many different forms. This is the reason there are multiple versions of the inference problem and the ways to obtain a valid solution are manyfold. According to a recent survey on the inference problem, it's very unlikely we will find one elegant universal solution for the inference problem of all the different variations in L-systems [3].

In this thesis, we will explore the current state of affairs regarding the syntactical inference problem for L-systems. Since there are many different kinds of L-systems, we decided to pick one specific type: deterministic, context-free L-systems, also called D0L-systems.

Since L-systems are often used to model complex processes in fields such as biology, we can use solutions to the syntactical inference problem as a new way of building models. If we would have a good solution to the inference problem, we would be able to build models from a small set of snapshots from reality. In this way, expert domain-knowledge is no longer necessary for obtaining a model from a set of observations. The building of a model would also become more cost-efficient and easier.

Before we can look into the definition of L-systems and its myriad of properties, we first have to take a step back and review the theoretical context. I will start this paper by placing L-systems in the broader context of formal language theory. After reviewing some preliminaires I will explain the most important different properties of L-system and give examples of their applications. After this we will explore the inference problem, and look specifically at the still open subproblems of the inference problem of D0L-systems. Finally, we will explain and analyse an algorithm that has been put forward recently as a solution to

this subproblem. We will consider this algorithm with a critical eye and check whether the authors deliver a good solution to the subproblem in question. We will finish with some suggestions to improve this algorithm.

2 Theoretical context

Since we will be speaking of concepts closely related to the theory of formal languages, we will now review some preliminairies. L-systems and the associated theory is especially related to formal grammars.

2.1 Formal Language theory

Formal language theory is the study of the form and behaviour of so-called formal languages: mathematical structured, artificial engineered languages, as opposed to human, informal, spoken languages. Ways to describe these formal languages include context-free grammars, context-sensitive grammars and regular expressions.

A formal language consists of words over an alphabet Σ. An alphabet Σ is a finite set of symbols. A words of length n over Σ is a string of symbols $u_1u_2u_3...u_n$, with every $u_i \in \Sigma$. The set of all possible words over Σ is denoted as Σ^*. Σ^* includes the empty string (a word made up of 0 symbols), denoted as λ. A way to describe a language is by defining a formal grammar. This grammar contains rules that tell us how a word in the language should be generated.

2.2 Grammars

Context-free grammars

A formal grammar is a triple (Σ, V, R), where Σ is an alphabet, V is a set of help-symbols, and R is a set of rules that define how the grammars generates words over Σ. In a context-free grammar, rules are of the form $u \to v$, where $u \in V$ and $v \in \Sigma \cup V^*$. Symbols from V are called non-terminals and are generally capitalized or uppercase. Symbols from Σ are also called terminals and are usually lowercase.

Take for instance the following grammar G with $\Sigma = \{a, b\}$ and $V = \{S, B\}$:

$$\begin{array}{lcl} S & \to & aB \\ B & \to & Bb \mid b \end{array}$$

Note that the rule

$$B \to Bb \mid b$$

is shorthand for

$$\begin{array}{lcl} B & \to & Bb \\ B & \to & b \end{array}$$

We produce valid words by following the rules in the grammar. Generally, we start with the symbol S. We then sequentially apply one of the rules from the grammar. We can stop rewriting when we have only terminals left in the string.

Below are some examples of productions of words using G:

$$
\begin{aligned}
S &\rightarrow aB \rightarrow ab \\
S &\rightarrow aB \rightarrow aBb \rightarrow abb \\
S &\rightarrow aB \rightarrow aBb \rightarrow aBbb \rightarrow abbb \\
S &\rightarrow aB \rightarrow aBb \rightarrow aBbb \rightarrow aBbbb \rightarrow ...
\end{aligned}
$$

We start with the symbol S. We then rewrite S to aB. We can now choose which rule we want to apply. We can either rewrite the B to Bb or to the terminal b. The last option gives us the first valid word: "ab". If we rewrite B to Bb, we can again either continue expanding the string with B, or make a new word "abb", and so on. It may be clear that this grammar describes the regular language $L(G)$ that consists of the words $\{ab, abb, abbb, ..., ab^n\}$.

Context-sensitive grammars

One of the drawback of this type grammar is that it's not possible to define more complex constraints, such as "Symbol X should rewrite to Y, but only when preceded by Z". Luckily, we can also define context-sensitive grammars. The difference with context-free grammars is that in context-sensitive grammars, rules have the form $u \rightarrow v$ with both u and $v \in \Sigma \cup V^*$. Now we can define more complex rules. Here's an example of a context-sensitive grammar G_2. $\Sigma = \{a, b, c\}$, $V = \{S, A\}$.

$$
\begin{aligned}
S &\rightarrow aAbAcAA \\
aA &\rightarrow aa \\
bA &\rightarrow bb \\
cA &\rightarrow cc
\end{aligned}
$$

This grammar can only rewrite the non-terminal A is there is either an a, b or c preceding it. The A will then be rewritten to two times the preceding character. Here is the derivation sequence for the production of the only word in $L(G_2)$. For clarity, the underlined characters are rewritten in the succeeding step.

$$\underline{S} \rightarrow \underline{aA}bAcAA \rightarrow aab\underline{A}cAA \rightarrow aabbc\underline{A}A \rightarrow aabbc\underline{cA} \rightarrow aabbccc$$

Let it be clear that in the second step of the derivation, we could have chosen to rewrite either $\underline{aA}bAcAA$, $aA\underline{bA}cAA$ or $aAbA\underline{cA}A$. The last A in the string can't be rewritten in this step, since it's preceded by another A. This means that none of the rules from the grammar are applicable. After we have rewritten the cA, the last A is preceded by a c. Only then are we able to apply rule 4.

The applications of formal grammars are endless. Some examples are syntax-definitions of programming languages, and representations of sets, algorithms and functions. L-systems are also a special kind of formal grammars. The biggest difference between 'normal' grammars and L-systems, is that instead of sequential application of rules, in L-system we apply all applicable rules in one step. This is called *parallel rewriting*.

3 L-systems

3.1 Origin and development

Aristid Lindenmayer introduced L-systems in 1968 [16] as a way of formalizing the development rules for plants and other biological organisms. The symbols from the grammar could be translated to real world objects or structures with the use of an interpretation function.

At first, the systems were only used for very simplified models of plants, bacteria and fungi. Over the years, a group of researchers [14] has expanded the use of L-systems beyond simplified models. Later, Lindenmayer and Prusinkiewicz introduced the use of brackets to denote branching systems. This extension of L-systems made it possible to model more than flat, simplified objects: now we could model branching systems such as trees, plants, but also branching structures in medicine such as arteries and dendrites.

With the evolution of computer imaging, L-systems are now also used to pseudo-randomly generate realistic images of trees, plants, and other structures. In the medical and biological sciences, L-systems are used to model and simulate the behaviour of cells, organs, organisms [21] and various processes associated with cell metabolism (such as the production of hormones in plants which greatly influence growth processes [15]).

At the moment there are numerous tools and frameworks online that can be used to interactively explore the possibilities of L-systems [2, 9, 17].

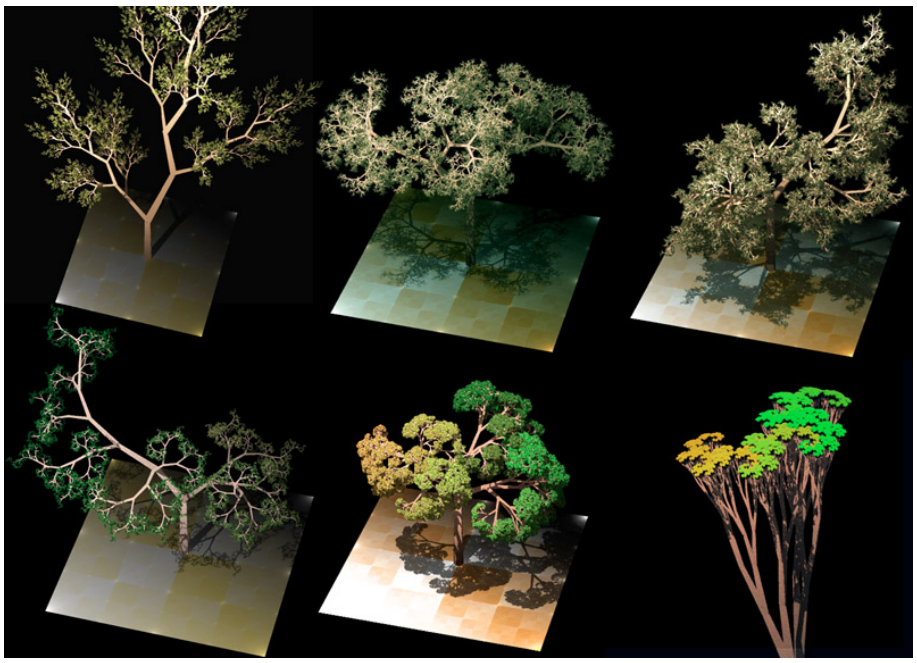


Figure 1: 3D plants generated with L-systems

3.2 What is an L-system?

Since we have clearly described how grammars work, we can now look at L-systems in the context of formal language theory. Basically, L-systems are grammars. The characteristic difference is that instead of choosing only one production rule when rewriting a word, we apply all possible rewriting rules in one step. This is called parallel rewriting.

If we want to define an L-system, we need to specify an alphabet, a set of rules and a starting point. The starting point is comparable to the starting symbol S of a formal grammar.

Take the following grammar that represents a very small subset of all valid sentences in English (adapted from [8]). Here the starting symbols S is denoted by the term "Sentence".

Grammar G_3

Sentence	$\rightarrow$	Subject FiniteVerb
Subject	$\rightarrow$	Article Noun
Article	$\rightarrow$	The
Noun	$\rightarrow$	cat | dog | mouse
FiniteVerb	$\rightarrow$	sleeps | eats | plays

A possible derivation sequence is the following:

Step			
1	Sentence	$\rightarrow$	Subject FiniteVerb
2		$\rightarrow$	Article Noun FiniteVerb
3		$\rightarrow$	The Noun FiniteVerb
4		$\rightarrow$	The dog FiniteVerb
5		$\rightarrow$	The dog sleeps

Note that it takes 5 rewriting steps to arrive at a valid sentence.

We can define the following L-system, based on the example above:

L-system L_1 based on G_3

The alphabet	Σ	$\{A, a, b, c\}$
Starting point	w_0	Sentence
Rewriting rules	rule 1	Sentence $\rightarrow$ Subject FiniteVerb
	rule 2	Subject $\rightarrow$ Article Noun
	rule 3	Article $\rightarrow$ The
	rule 4	Noun $\rightarrow$ dog
	rule 5	FiniteVerb $\rightarrow$ sleeps

The derivation sequence to obtain the same sentence is as follows for L_1:

Step			
1	Sentence	$\rightarrow$	Subject FiniteVerb
2		$\rightarrow$	Article Noun sleeps
3		$\rightarrow$	The dog sleeps

Note that it takes only 3 rewriting steps to arrive at a valid sentence.

Here we can clearly see the difference between a normal grammar and an L-system. Whereas in the grammar example, we had to choose which one of the two applicable rules we would apply first, we can now rewrite 'Article Noun' in one step to 'The dog'.

3.3 Formal definition

There are various definitions of L-systems . Here we will use the one defined by Lindenmayer in [16].

An L-system is a triple (Σ, P, w_0) with an alphabet Σ, a grammar or set of production rules P and a starting point, w_0. The production rules are of the form $A \to w$, with $A \in \Sigma$ and $w \in \Sigma^*$.

Note that there is a difference with the definition of a formal grammar we encountered earlier. L-systems have only one symbol from the alphabet on the left side of the rule, whereas rules in standard grammars can have multiple. Secondly, there is no difference between terminals and non-terminals. For all symbols $\sigma \in \Sigma$ that don't have a rule, we implicitly take the rule $\sigma \to \sigma$.

3.4 Turtle interpretation of strings

Since L-systems are used to model all kinds of objects, we want to have some kind of graphical interpretation for strings generated by an L-system. Many interpretations have been suggested [16]. One of the most used interpretations is the *turtle interpretation of strings*, which is closely linked to Turtle Geometry, as is used in the programming language LOGO [1].

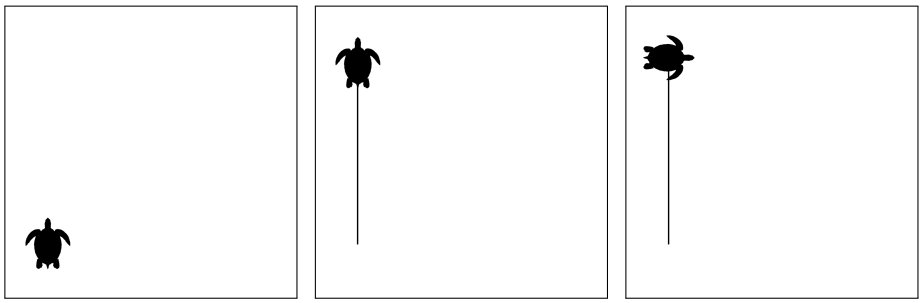
Turtle geometry is a notion from computer graphics theory that refers to drawings figures by means of "commanding" a turtle in a Carthesian plane. We define the *state* of the turtle as a triplet (x, y, α), with the x-coordinate, the y-coordinate and the angle α to record the direction in which the turtle is facing.

The turtle starts at a certain point with a certain orientation. Given an angle δ and a step size d, we can command it to move forward, turn left or right, with the following commands: [1]

- F Move forward a step of length d.
  The state of the turtle changes from (x, y, α) to (x', y', α),
  where $x' = x + d \cos \alpha$ and $y' = y + d \sin \alpha$.
- $+$ Turn left with angle δ.
- $-$ Turn right with angle δ.

The commands for the turtle can be edited and ammended as desired depending on the application. We could, for instance, define commands for drawings squares, circles, etc. The three commands mentioned above are the most used in the literature.

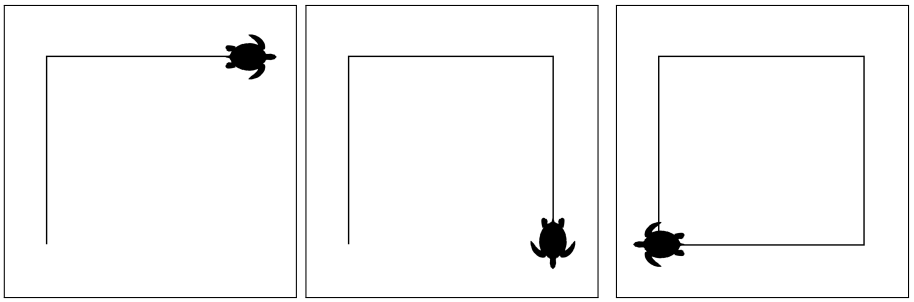
In the turtle interpretation of strings, each symbol or sequence of symbols in a string is interpreted as a command to the turtle. Given a string s, a starting



(a) The turtle starts facing north.

(b) The turtle draws a line.

(c) The turtle turns right by 90 degrees.



(d) The turtle draws another line.

(e) The turtle draws a third line.

(f) The turtle completes the square.

Figure 2: The turtle interpretation of the string F-F-F-F

point, an angle and a step size, we can draw a the turtle interpretation of s as a figure. A simple example:

1. We want to interpret the string F-F-F-F.
2. We take an angle of 90° and a step size of 1.
3. Starting point of the turtle: (0,0).
   Orientation of the turtle: the turtle is facing north.
4. The turtle starts drawing a square:

Symbol	The following happens:
F	The turtle moves forward (and draws a line).
$-$	The turtle turns right (by 90 degrees).
F	The turtle moves forward.
$-$	The turtle turns right.
F	The turtle moves forward.
$-$	The turtle turns right.
F	The turtle moves forward.

5. The turtle has drawn a square. We now have the turtle interpretation of the string F-F-F-F.

3.5 Variations

Because L-systems have been used to model various objects, structures and processes, some authors have taken to extend the basic notion of L-systems. In this section I will explain the most important variations.

Deterministic and stochastic L-systems

An L-system is deterministic if there is exactly one production rule for each symbol $s \in \Sigma$. All elements of a derivation sequence then adhere to the unique successor condition (as described in [7]). A L-system is stochastic if it is nondeterministic and the production rule chosen is dependent on some parameter in the interval [0,1]. The chance that a derivation rule is chosen is depicted above the arrow. A very simple but realistic example of a stochastic L-system is the following:

Stochastic L-system *Garden*

w_0 : SPACE
p_1 : SPACE $\xrightarrow{0.15}$ PLANT
p_2 : SPACE $\xrightarrow{0.80}$ WEEDS
p_3 : SPACE $\xrightarrow{0.05}$ FLOWER

Propagation

An L-system is non-propagating if it contains rules of the form $s \to \lambda$, with $s \in \Sigma$. It follows that a propagating L-system has no rule $s \to \lambda$. This way, symbols in a word can not simply disappear in a rewriting step. This means that in a derivation sequence of a propagating L-system, the length of every word in the sequence is the same or larger than its predecessor.

Context-free and Context-sensitive

Just as there are context-free grammars and context-sensitive grammars, there exist context-free and context-sensitive Lindenmayer-systems. Context-free L-systems, or 0L-systems, have production rules of the form $s \to w$, with $s \in \Sigma$ and $w \in \Sigma^*$.

In the literature we also encounter 1L-systems (unidirectional L-systems) and 2L-systems (bidirectional L-systems). The n in nL-systems refers to the number of neighbour-symbols that have any influence on the derivation process. This extension of L-systems was introduced to be able to take the "state" of neighbouring cells into account when modelling biological structures such as plants and fungi. Prusinkiewicz used these systems in [15] to model cell metabolism processes such as the transport of enzymes from one cell to the next.

Before we can starting working with context-sensitive L-systems, we have to extend the definition of an L-system. The production rules in the set P change from

$$s \to w, \text{ with } s \in \Sigma \text{ and } w \in \Sigma^*$$

to

$$(l, s, r) \to w, \text{ with } l, s, r \in \Sigma \text{ and } w \in \Sigma^*$$

With this extension we can set specific rules for s depending on the characters of the left-neighbour-symbol l and the right-neighbour-cell r. We can take a context-free L-systems and change it into a context-sensitive L-system by substituting all the rules

$$s \to w$$

by the rules

$$(a, s, b) \to w \text{ for all } a, b \text{ in } \Sigma.$$

In 2L-systems, both the left and right-neighbour cells exert influence over the rewriting. This means that a symbol s can only be rewritten to w, if its left and right neighbours are adhering to the constraints as stated by the production rules. For example, given a rule $(a, s, b) \to w$, the symbol s can only be rewritten to w if its left-neighbour is a and its right neighbour is b.

In 1L-systems, or unidirectional system, the derivation rule chosen is only dependent on the left or right-neighbouring-symbol. This means that if an L-system is unidirectional, all production rules have the following property:

(a) If $(d, s, e) \to w$ then $(d, s, f) \to w$ for all $f \in \Sigma$; or,

(b) if $(e, s, d) \to w$ then $(f, s, d) \to w$ for all $f \in \Sigma$;

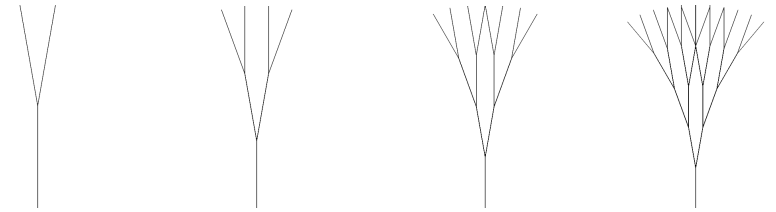
Bracketed L-systems

A string of symbols between brackets denotes a branching structure. In this way it has become possible to use L-systems to model branching systems and organisms, such as trees and complex plant-structures. In the turtle interpretation of strings, a bracketed part of a word means that everything contained in the bracket is a seperate branching structure.

The idea is that at the start of a bracket, the turtle continues the route described within the brackets. At the closing bracket, the turtle is propelled back towards the point and orientation where he was at the opening bracket, and the remaining symbols of the word are interpreted from the point where the turtle is now standing again. Keep in mind that in this interpretation, it is possible for the turtle to overwrite the lines created by previous branches.

Simple example of a bracketed L-System

F → F[-F][+F]
angle = 5



(a) Iteration 1 (b) Iteration 2 (c) Iteration 3 (d) Iteration 4

Figure 3: Four iterations of the bracketed L-system

3.6 Applications

Biology

Leitner et al used stochastic L-systems in Matlab to model the interaction between soil and plant roots. [11]

Prusinkiewicz and Lane showed the use of L-systems in the study of morphology in [15]. Models were built to predict heterocyst differentiation in *Anabaena* bacteria and growing patterns in ivy leaves. By modeling the exchange of auxins, plant hormones that eventually determine the pattern of veins and the form of the leaf, they could build a model of the molecular details of ivy leaf development.

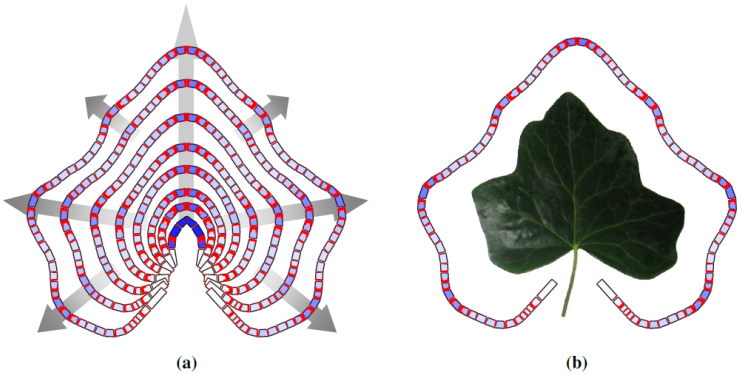


Figure 4: (a) An L-system model of ivy leaf development. (b) Comparison of the final shape generated by the L-system with a real ivy leaf.
Image obtained from [15].

Since the morphology of the fungus *Alternaria* is very complex, Taralova et al have made a model with L-systems [21] that can be used to explore the morphology of Alternaria under different circumstances. The model can also generate model groups (depending on different inputs) that can be compared to microscopic images to find parameters for species-specific models. L-systems are here used to facilitate research that would otherwise be too time-consuming to undertake.

Figure 5: 3D Trees generated by the blender plugin [4] based on L-systems

Computer science

Lima de Campos et al use L-systems as a representation of growing patterns in brainstructures, to optimize the machine learning concept of artificial neural networks [5].

Since we can use the turtle interpretation of strings for drawing 2D and 3D images, L-systems can also be used for various applications in computer graphics. For instance, it is possible to download a plugin for the 3D-software Blender, that generates vegetation in 3D by means of stochastic L-systems [4].

L-systems can also be used to generate fractals, significantly self-similar, space-filling curves such as the Peano-curve.

Medical sciences

Mulchandani constructed a stochastic L-system [12] for the generation of axons and dendrites. This way he obtained synthetic neurons that could be compared to manually-traced neurons. The L-system contributes to the development of neuron databases: it allows us to estimate the new information contributed by the additional neuron morphological measurements.

In Jelinek et al [9] we find a new take on the modeling of growth patterns we saw earlier in [15]. The research team presents a new model for neuron growth using L-systems and a tool called Micromod, a web-based-tool for the generation of neuron models based on some parameters. In [2] we find another model generation toolkit based on L-systems. The tool L-neuron contains ways of generating topolical models of dendrites that can be used for analytical research and 3D-representation.

In 2001 Zamir investigated the possibility of using parametric L-systems [22] to model arterial branching systems. The author concludes that we do not yet have enough information regarding the range of variability in arteries to include all properties in L-systems. We can only model some properties of the cardiac system with L-systems.

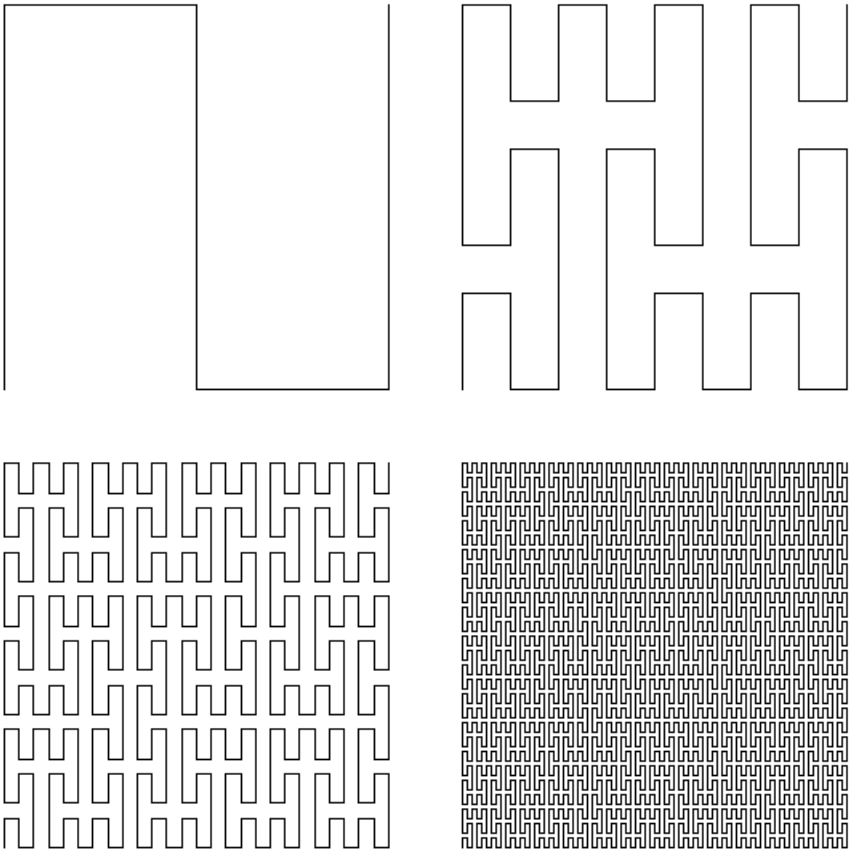


Figure 6: Four iterations of the peano-curve

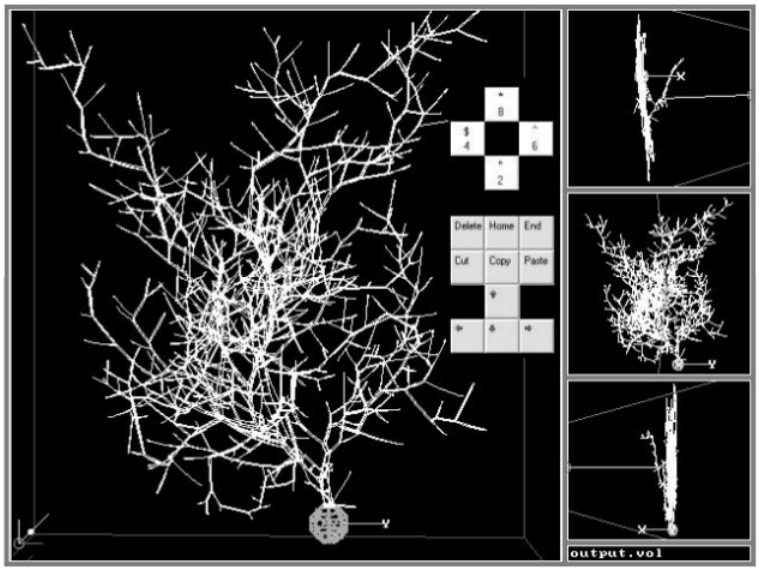


Figure 7: A Purkinje cell [2] generated by L-neuron.

Because some processes in the ventrikels of the heart cannot be observed in living organisms, we need a different way of obtaining a good biophysical model. In two recent articles by Sebastian et al on medical imaging we encounter L-systems again as a way of modeling the Cardiac Conduction System (CCS), from samples obtained from calf and lamb tissue. [19, 20] The model is then used to construct a 3D model of the CCS, given a ventricular anatomy.

4 The inference problem

4.1 Problem statement

Basically, the inference problem for L-systems is this: given a derivation sequence of an L-system, can we reconstruct the L-system that generated it? The inference problem is a lot like other grammatical inference problems. We could describe it as the discovery of structures from examples that have presumably been generated by the the same structure [3]. In this case the structures are L-systems.

4.1.1 Original problem definition

We will use the description of the inference problem as coined by Feliciangeli and Herman in [7]. Since the way we can solve this problem is dependent on the input sequence, the authors distinguish three subproblems of the inference problem:

1. Give an algorithm which decides for any finite set of sequences of strings whether or not there exists a grammar of a certain type (one of 12 possible types) such that each of the sequences is
   (a) a part of a derivation by a grammar; or,
   (b) a regular sampled part of a derivation by the grammar; or,
   (c) a randomly sampled part of a derivation by the grammar.
2. Give an algorithm which, whenever the algorithm in (1) produces a positive answer, will produce an appropriate grammar.

4.1.2 Input types

As stated in the problem definition above, the algorithm we can use to solve the inference problem is dependent on the dataset that we use as input. Generally, the input is a set of sequences of strings. The 'distance' between strings, the number of derivation steps between two strings, plays also a vital role in the hardness of the inference problem. The inference problem deals with three kinds of derivation sequences:

(a) a part of derivation sequences. This means that the input is of the the form

$$s_i \to s_{i+1} \to s_{i+2} \to ... \to s_{i+n}.$$

(b) a regular sampled part of a derivation sequences, in other words

$$s_i \to^n s_{i+n} \to^n s_{i+2n} \to^n ... \to^n s_{i+mn}.$$

(c) a randomly sampled part of a derivation sequence:

$$s_i \to^a s_{i+a} \to^b s_{i+a+b} \to^c ... \to^z s_{i+a+b+..+z}$$

Bigger distances are more challenging and sometimes the derivation distance between strings is unknown. Additionally, the distance between strings in a sequence can vary. We need algorithms that can deal with all kinds of datasets.

4.1.3 Targets

Feliciangeli and Herman also made the distinction between 12 kinds of grammars that we can choose as *target*. The target is the kinds of L-system we want to infer from the derivation sequence. All targets have a combination of properties, as described in section 3.5. We distinguish the various types of grammars. Each type has a combination of the following properties:

1. the grammar is either non-deterministic or deterministic;
2. informationless, unidirectional or bidirectional;
3. propagating or non-propagating.

The properties of the target determine the hardness of the inference problem. It also makes a big difference in the choices we have for picking a certain algorithm. For instance, the inference problem for propagating deterministic 0-sided L-systems (PD0L-systems) is on a complete different level of hardness than the inference problem for non-deterministic 2-sided L-systems. The PD0L-problem has a search space of

$$|T| \cdot |\Sigma^*|$$

possibilities, where $T \subseteq \Sigma$ is the set of non-terminals in the alphabet. Most of the time we can dramatically reduce that search space by taking into account some properties of the input sequence.

The non-deterministic 2-sides L-system on the other hand, has a much larger search space. Since every non-terminal is allowed to have more than one rule, the possibilities are endless. The maximum size of the search space for a N2L-system is

$$|N| \cdot |\Sigma| \cdot |T| \cdot |\Sigma| \cdot |\Sigma^*|$$

Thus, the inference problem of L-systems can be divided in 36 subproblems, one for each combination of the twelve targets and three types of input.

4.2 Closed and open problems

In [7] Feliciangeli and Herman prove the existence of a solution to 30 of the 36 subproblems. They do this by offering constructive proofs [3] of the decidability of existence of an inference algorithms for each input type. In the table below we can find a schematic overview of the results of Feliciangeli and Herman.

Type of L-system		Deterministic		Non-deterministic	
		Propagating	Non-propagating	Propagating	Non-propagating
x=0	A	Dec	Dec	Dec	Dec
	B	Dec	Dec	Dec	Dec
	C	Open	Open	Dec	Dec
x=1	A	Dec	Dec	Dec	Dec
	B	Open	Open	Dec	Dec
	C	Open	Open	Dec	Dec
x=2	A	Dec	Dec	Dec	Dec
	B	Dec	Dec	Dec	Dec
	C	Dec	Dec	Dec	Dec

'Dec' indicates that the problem in question is decided. x = n refers to the context-sensitivity of the L-system. If $n = 0$ the system is informationless, if $x = 1$ the system is unidirectional and if $x = 2$ the system is bidirectional. The A, B and C refer respectively to (a),(b) and (c) in the definition of the inference problem as stated above in section 4.1.1.

In the survey from 2010 by Ben-Naoum [3] the author shows that some of the open problems mentioned by Feliciangeli and Herman have recently been solved. However the algorithms that provide us with new solutions to the inference problem often need more information than just the target and a derivation sequence. Some algorithms offer only a partial solution to a subproblem. Furthermore, all algorithms have their own strong and weak points. For a short overview, see section 5.1.

4.3 Scientific relevance of the inference problem

Providing solutions to the inference problem is interesting for several reasons. Below we will discuss the importance of the problem in the main areas of application.

Computer science

From a computer science perspective, the inference problem for L-systems is interesting because of its similarity to other grammatical induction problems. L-systems are similar to grammars with parallel rewriting rules. A grammar can be an efficient tool to summarize the knowledge we have about a large set of strings (a language) and finding such a grammar for a small set of example strings is a challenge that can be linked to the fields of information retrieval, modeling and machine learning. The inference problem addressed some of the common problems in learning. It can also give us useful information about the learnability of *targets*.

Biology and the life sciences

Since L-systems were originally made for modeling organisms in biology, let us take a look at the way solution to the inference problem can contribute to this

field.

In Biology, one way to build a model for an organism is to work bottom up: starting at the way neighbouring cells interact, we move up until we have a model that includes all relevant processes. For this way of modeling, a great deal of domain knowledge (from molecular science to chemistry and physics) is required. The process is time-consuming and prone to mistakes.

Another way to build a model is to observe an organism over time. If we can formalize the observations, we can use solutions to the inference problems to generate rules that explain these developments. This way of modeling requires less domain knowledge, although we do need a sound method of formalizing observations.

If we can find a proper string representation for scientific observations, we can use solutions to the inference problem to infer a grammar that can serve as a model for the observed processes. Depending on the kind of model we want, we choose a target, and depending on the type of observations we have, we choose an input type.

5 Solutions to the DC0 inference problem

In this section we will discuss in short a few solutions to the open inference problem for deterministic context-free L-systems. The inference problem for D0L-systems is divided into three subproblems: DA0, DB0 and DC0.

DA0 problem : the inference problem given a part of a derivation sequence.

DB0 problem : the inference problem given a sampled part of a derivation sequence. Between all strings in the sequence, there are n derivation steps. We have no knowledge about the size of n, only that it's constant.

DC0 problem : the inference problem given a randomly sampled part of a derivation sequence. Between all strings in the sequence, there is an unknown random number of derivation steps.

Note that the A, B and C refer to the three subproblems as described in the original problem statement, see section 4.1.1.

In this section, we are specifically looking at the subproblem for randomly sampled derivation sequences. We will refer to this specific subproblem as the DC0 inference problem. Afterwards, we will analyse a new solution from recent literature and give an elaborated example to illustrate how the algorithm works. Finally we will discuss some proposed changes to the algorithm and possibilities for further research.

5.1 Earlier solutions to the DC0 inference problem

One of the first solutions to the DC0 problem was given by Doucet [6]. He gives an algorithm for inferring D0L-system given a 'scattered word sequence'. The algorithm is based on algebraic operations on Parikh vectors. However, Doucet assumes that both the alphabet and the word-rank-numbers (their index in the derivation sequence) are known. According to the definition of Feliciangeli and Herman, we only have the derivation sequence as input. This algorithm only works with more *a priori* knowledge than stated in the original problem definition.

In [13], Nevill-Manning and Witten infer a D0L-system from a single string, by following an algorithm named SEQUITUR that reminds us of data-compression. Since this algorithm only takes one string as input, we can interpret algorithm as a solution to the DA0, DB0 and DC0 inference problems. Recursively, the algorithm replaces repeated substrings in the input string by L-system rules that generate that same substring. The final number of rules is reduced by the following two constraints:

1. No pair of adjacent symbols appears more than once in the D0L-system.
2. Every rule is used more than once.

The algorithm showed good results on large-sized DNA-sequences. Besides the heavy memory usage, one drawback was that the algorithm was not able to deal with brackets. Thus, any grammar it produced would be unfit for modeling

problems in biology. A new version of SEQUITUR was made, but in the process the algorithm lost its lineair time complexity [3].

In 1997 John Koza [10] introducted another way of inferring L-system, this time by using genetic programming. He only demonstrates his algorithm with only one example, so we can't be sure we will always arrive at a good solution within reasonable time. Genetic algorithms are known for their high memory usage and their inefficiency. However, genetic algorithms can also lead to very good results with difficult problems.

5.2 A new partial solution to the DC0 inference problem

In [18] Santos and Coelho coin a solution to the *inverse problem* of L-systems, which can be interpreted as a partial solution to the third inference problem of D0L-systems. The inverse problem of L-systems is the problem that deals with L-system induction given some productions of that L-system. Note that the inference problem of L-systems is a subproblem of the inverse problem of L-systems.

As mentioned before, the third inference problem deals with an input of a randomly sampled derivation sequence. Since the algorithm of Santos and Coelho takes only one string (as opposed to a complete sample of a derivation sequence), we can interpret the input as a randomly sampled derivation sequence of length 1.

Santos and Coelho take a string and derive a L-system that possibly produced it. They do this by means of a combination of pattern-searching and simple mathematical operations. This method is fairly effective, but there are some side notes to be made. Before we start analysing the effectiveness and complexity of their solution, let us look at the algorithm.

An implementation of the algorithm as discussed in "Obtaining L-system rules from strings" is provided in the appendix.

5.2.1 Algorithm

In this subsection we will provide the reader with an elaborated example of how the algorithm processes the input string to arrive at a set of possible rules. The pseudocode of the algorithm can be found in figure 8.

We start with a string s over an alphabet Σ. This is the only input the algorithm needs. For $\Sigma = \{\mathrm{F}, +, -\}$, consider

$$s = \mathrm{F+F+++F+F++++}$$

We count F_s, the number of times we encounter an F in s. The algorithm expects that $F_s > 1$ (more on this in section 5.2.2). In our example,

$$F_s = 4$$

We then try to find a combination of F_r and n, where F_r is the number of F's in the rule of the L-system we are trying to infer and n is the iteration of the derivation in which s was produced. The following equation should hold

```
1  Function InverseProblem(w:string):string
2  begin
3      Ft = CountSimbol('F',w)
4      St = length(w)-Ft
5      if Ft > 1 then
6          for Fq = 2 to int(sqrt(Ft)) do
7              n = ln(Ft)/ln(Fq)
8              if isInteger(n) then
9                  str = w
10                 oldpat = Slice(w,Ft,Fq)
11                 if Replace(str,oldpatt,'F') == Fq then
12                     rule = TerminalClear(str,n)
13                     if isPossible(rule,Fq,n,St) then
14                         return rule
15                     end if
16                 end if
17             end if
18         end for
19     end if
20     rule = w
21 End
```

Figure 8: The Pascal code from the paper bij Santos and Coelho [18]

$$F_s = F_r^n$$

We find these combinations of F_r and n by brute-forcing all possibilities for F_r between 2 and $\sqrt{F_s}$. Both F_r and n should be integers. We search an F_r and n such that

$$\log_{F_r} F_s = \frac{\ln F_s}{\ln F_r} = n$$

For our example, we find the following combination of F_r and n:

$$(F_r, n) = (2, 2) \text{ because } \frac{\ln 4}{\ln 2} = 2$$

Now we try to divide our input string in F_r equal parts with k occurences of F each. Here $k = F_s/F_r$. We start searching for $F_r = 2$ substrings of s that go from F_0 to F_k and F_{k+1} to F_s. The substrings should be the same. This is done by the Slice-function, see line 10 in figure 8.

$$s_{sliced} = \boxed{\text{F+F}}\text{+++}\boxed{\text{F+F}}\text{++++}$$

We now substitue the pattern we found in the previous step with F. This operation, as implemented in the Replace()-function, gives us the following string:

$$s_{replaced} = \boxed{\text{F}}\text{+++}\boxed{\text{F}}\text{++++}$$

If $F_{s_{replaced}} = F_r$, then we can try to make a rule out of this new string. But there's one last post-processing step we must do. As we can see in the example,

it is possible that some terminal-symbols have accumulated in $s_{replaced}$. The TerminalClean-function (line 11 in figure 8) processes the string such that we will end up with the right amount of terminals in the rule. We count the *leading* terminals T_l and the *trailing* terminals T_r. If we have the string

$$\text{F+++F++++}$$

any terminals in the head of the string are leading characters and any terminals in the tail are trailing characters. Since the string starts with a non-terminal, there are no leading characters we have to delete. The four plus signs in the tail of the string are the trailing characters. This means that in our example

$$T_l = 0 \text{ and } T_r = 4$$

We then determine how many terminals should be removed from s to make a fitting rule. R_L tells us how many terminals should be removed before each occurrence of F, and R_r gives us how many terminals should be removed after each occurence of F.

$$R_l = \frac{T_l}{n} * (n-1)$$

$$R_r = \frac{T_r}{n} * (n-1)$$

In our example, $n = 2$, which gives us

$$R_l = \frac{0}{2} * 1 = 0$$

$$R_r = \frac{4}{2} * 1 = 2$$

A R_r of 2 means that 2 *trailing* characters will be removed after each occurence of F. The following characters will be deleted from s:

$$s_{replaced} = \text{F}\underline{\text{++}}\text{+F}\underline{\text{++}}\text{++}$$

which gives us

$$s_{terminalcleaned} = \text{F+F++}$$

leading to the rule

$$\text{F} \rightarrow \text{F+FF++}$$

The only thing left to do, is to check whether the inferred rule with is a *good* rule. In other words, will the axiom F together with the inferred rule give us the string s in n derivation steps?

The authors check this by counting the number of terminals T_s and the number of F's in s and checking this with the nth iteration derivation-sequence made by the inferred rule and the axiom F. We suggest that we check the derivation more in depth, by generating the nth iteration.

axiom : F
rule : F → F+F++

1	F	→	F+F++
2		→	F+F+++F+F++++

Indeed, the derivation verifies that F $\rightarrow$ F+F++ can produce the input string in 2 steps. F $\rightarrow$ F+F++ is a valid rule.

5.2.2 Analysis

The algorithms from Santos and Coelho has a few strong points compared to the other discussed algorithms from [3].

1. Compared to the other algorithms for the DC0 problem, this algorithm can work without knowing the full alphabet. The only thing we need to know is the symbol of the only non-terminal. This non-terminal we can directly obtain from the input string. One could say that the algorithm needs to know the alphabet for the TerminalClean()-operation, but since we only have one rule (and thus one non-terminal), any symbol that's not the non-terminal is automatically a terminal and should be taken into account when counting the trailing and leading characters in the input string.
2. The algorithm only needs one input string. This is both a strong and weak point. Since we only have one input string, we don't have to delve into the tedious task of inferring things from intervals in the derivation sequence we do not know. The only thing we do need, is to find the iteration n, which can simply be obtaining by brute-forcing all possibilities, which is fairly easy. On the other hand, the algorithm will be prone to overfitting. Because of this single input string, we are badly limited in the L-system we can infer. Combined with the property of L-system that are limited to one rule, we can only infer rather trivial L-systems.
3. Compared to the other proposed algorithms, this algorithm only uses very basic operations. It will be relatively fast and hardly needs any memory.
4. The authors have included an L-system generator in their code, that generated random input strings with which they could validate the algorithm. In all cases, the algorithm produced an appropriate L-system.

Besides these strong points, the algorithm by Santos and Coelho has some serious restrictions, that should also be taken into account.

The algorithm by Santos and Coelho has the following restrictions:

1. We have to assume that the input string is derived from an L-system with only one rule. This is a very limiting constraint. For example, to generate the Koch-island, a well-known and much-used example in L-systems, we need two rules. By limiting the target to one-rule L-system, we exclude a large and interesting subset of the set of all possible L-systems.
2. Furthermore, the authors assume that the input string is derived from an axiom w_0 which only contains one symbol, namely the only non-terminal from Σ. This is also a strong limitation. Even if an L-system adheres to the one-rule-constraint, it's not necessarily inferable with this algorithm. For example, if we have an L-system L with the rule:

$$\text{F} \rightarrow \text{F+F}$$

then the algorithm can only infer L if and only if it has a string from the following sequence as input:

$$\text{F} \rightarrow \text{F+F} \rightarrow \text{F+F+F+F} \rightarrow \text{F+F+F+F+F+F+F+F} \rightarrow \ldots$$

3. The authors don't take into account rules that contain only one non-terminal and some terminals. Think for instance of an L-system with the rule

   $$\text{F} \rightarrow \text{F+}$$

   The algorithm won't work for this L-system, since it assumes that all L-system rules $F \rightarrow x$ have an x with more than one non-terminal. This is also the reason why the the algorithm (line 5 of 8) postulates that F_s should be strictly larger than 1. But as we can see in this derivation sequence

   $$\text{F} \rightarrow \text{F+} \rightarrow \text{F++} \rightarrow \text{F+++} \rightarrow \ldots \rightarrow \text{F+}^n$$

   the authors exclude a large subset of the one-rule L-systems with this constraint.

4. After we have derived a possible rule for the L-system based on the input string, we have to verify our rule. The authors apply the derived rule on their standard axiom F and verify their rule by counting the number of terminals and non-terminals the generated string. If these numbers accord with the numbers of non-terminals and terminals in the input-string, they see their rule as valid. It would be better if the algorithm would check the rules in more detail, since these kinds of algorithms are prone to programming mistakes. We should be sure that the input string $i_1 i_2 i_3 ... i_m$ is an exact copy of the derived string $d_1 d_2 d_3 ... d_k$.

5. The inference problem as specified in 4.1.1 asks for an algorithm that first checks whether it is possible to infer an L-system based on the input string(s), and secondly an algorithm that provides the L-system if there is one. However, because this algorithm was meant to provide a solution to the *inverse problem* and not the inference problem, the termination behaviour of the algorithm does not exactly correspond to the demands of the inference problem. The inference problem definition states, that an algorithm should either output 'False' (in the case that it can't infer L-system for the input sequence of strings), or give an L-system as output. This algorithm, on the other hand, always gives a rule as output, even in the case that the algorithm wasn't able to infer one! In that case, the algorithms outputs the trivial solution

   $$\text{F} \rightarrow i, \text{ where } i \text{ is the input string}$$

   . We should keep in mind that, when the algorithm terminates, this means that it was not possible to find a L-system

   (a) that is not the trivial L-system with the rule $\text{F} \rightarrow i$, where i is the input string

   (b) that is deterministic and informationless

   (c) with only one rule

(d) that contains more than one non-terminal in the tail of the rule

(e) that produced the input string by applying one rule to an axiom with only one symbol

This means that the discussed algorithm only provides us with a partial solution to the DCO-problem.

5.2.3 Improvements

It may be clear that the solution can only work for a very small subset of all D0L-systems and a very small subset of input strings. We think the algorithm can be greatly improved in the following way:

1. Firstly, we should improve the way the algorithm verifies possible rules. The original algorithm only compares the number of terminals and non-terminals in the input string and the derived verification string. An improved verification would check if the input string $i = i_1 i_2 i_3 ... i_m$ is an exact copy of derived string $d = d_1 d_2 d_3 ... d_k$ we obtained by applying the inferred rule n times on the standard axiom F. We should check this by determining whether $m = k$, $i_1 = d_1$, $i_2 = d_2$, and so on.
2. We also want to propose a small change to make the algorithm suitable for a slightly larger set of input strings. The current algorithm doesn't live up to its promise of providing a inference for all one-rule D0L-systems. It should be expanded for rules with more than one character in the axiom.
3. Another point of point of improvement is the inability to deal with rules that contain only one non-terminal in the tail.
4. We should devise a criterium or a weight-function for the algorithm so it can decide which rule from the generated rule set is the best. For example, we could choose the generated rule with the shortest tail.

5.2.4 Python code

In the appendix, we can find an implementation of the discussed algorithm. The implementation has been made to clear up some ambiguities in the paper, especially the functionality of the TerminalClean-operation. There are also few differences between the implementation in the appendix and the original algorithm from figure 8. All line references in this section refer to the Python code in the appendix, unless stated otherwise.

1. The description of the function TerminalClean() in the original paper was very ambiguous. After working out some examples, the informal description from the original paper became clear. We hope that the Python code and the elaborated example from section 5.2.1 provide a better description of how this function works.
2. Furthermore, the Python code sacrifices efficiency for completeness. We have chosen to calculate all possible combinations of (F_r,n) beforehand.

This is done with the function Bftuples (lines 38 and 142), which returns a list of tuples (F_r, n) such that $F_s = F_r^n$.

3. As a result, our algorithm tries to find *all* possible L-systems, instead of just one. The algorithm iterates over all tuples generated by Bftuples (line 142) and checks whether they yield a valid rule (line 147). Therefore, the output of the algorithm is not just one rule, but a set of possible rules. In line 141 we add the solution F $\rightarrow i$, where i is the input string. This way the algorithm always outputs the trivial solution.

4. The implementation used a stricter way of checking whether a rule is valid, as discussed in section 5.2.3. It uses the substitute-function from the regular expressions library. In line 126, we recursively substitute all occurences of F in a derivation string with the rule. The derivation string starts with the axiom 'F'. After the derivation process, we check whether the derived string is *identical* to the input string (line 146). Only then, the rule is declared valid and added to the rule set (line 147).

5. A final difference is the working of the Replace-function. In the original algorithm, the replace-function changes the original string *in situ* and returns only an integer: the number of substitutions made by the function. As such, it can be used in the if-statement from line 11 in the pseudocode. In the implementation, we use two lines instead of one. The function Replace returns the changed string (line 144), and we use the countF-function (line 145) to check the number of replacements.

6 Conclusion

6.1 Summary

In the first chapter of this thesis, we have put Lindenmayer-systems into the context of formal language theory. We have seen the differences and similarities between L-systems and grammars. We have explored the various properties of L-systems that we encounter in the literature, such as branching systems and stochastic systems, and their applications.

After stating the general definition of the inference problem for L-systems, we have taken a closer look at the inference problems for deterministic, context-free L-systems. We have reviewed some partial solutions from the literature for one of the open subproblems, the DC0 problem. This problem deals with inferring an D0L-system from a randomly sampled part of a derivation sequence.

Finally, we have elaborated on a recently coined solution. Our specific algorithm deals with a random sampled input sequence of length 1, which can be used to solve the DC0-subproblem. After analysing the algorithm, we came to the conclusion that the algorithm can only deal with very specific kinds of L-systems, namely one-rule D0L-system with more than one terminal in the tail. Secondly, the algorithm can only infer rules if the input string was generated by applying one rule to an axiom "F".

We have provided an annotated implementation of the algorithm that uses an improved way of validating rules. Furthermore, our implementation generates all possible rules, including the trivial rule, instead of just one rule.

The algorithm can probably be expanded so it can infer a broader range of L-systems. It seems that there still no good solution to the DC0 problem.

6.2 Future research

The things discussed in previous sections lead us to the following further research questions:

1. Can the algorithm as discussed in 5.2.1 be improved to deal with
   (a) D0L-systems with more than one rule?
   (b) D0L-systems with rules that contain only one terminal (for example $F \rightarrow F+$)?
   (c) input strings that were generated from an axiom that is *not* "F"?
2. Can we find a better solution for the DC0-problem by reinterpreting a solution to the inverse problem of L-systems?

References

[1] Harold Abelson and Andrea A Di Sessa. *Turtle geometry: The computer as a medium for exploring mathematics.* the MIT Press, 1986.

[2] Giorgio A Ascoli and Jeffrey L Krichmar. L-neuron: a modeling tool for the efficient generation and parsimonious description of dendritic morphology. *Neurocomputing*, 32:1003–1011, 2000.

[3] Farah Ben-Naoum. A survey on l-system inference, 2009.

[4] C. Dawson. Liquidweb: L-systems. a plugin for blender. `http://lsystem.liquidweb.co.nz/`. [Online; accessed 12-June-2013].

[5] L.M.L. de Campos, M. Roisenberg, and R.C.L. de Oliveira. Automatic design of neural networks with l-systems and genetic algorithms - a biologically inspired methodology. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*, pages 1199–1206, 2011.

[6] P.G. Doucet. The syntactic inference problem for d0l-sequences. *Lecture Notes in Computer Science*, 15, 1974.

[7] Horacio Feliciangeli and Gabor T. Herman. Algorithms for producing grammars from sample derivations: a common problem of formal language theory and developmental biology. *Journal of Computer and System Sciences*, 7(1):97–118, 1973.

[8] H. Geuvers. Formeel denken: Lecture notes. `http://www.cs.ru.nl/~freek/courses/fd-2011/public/fd.pdf`, 2011. [Online; accessed 07-June-2013].

[9] Herbert Jelinek, Audrey Karperien, David Cornforth, Roberto Marcondes Cesar Junior, and Jorge de Jesus Gomes Leandro. Micromod—an l-systems approach to neuron modelling. In *Proceedings of the Sixth Australasia-Japan Joint Workshop on Intelligent and Evolutionary Systems, Australian National University, Canberra*, pages 156–163, 2002.

[10] John R. Koza. Discovery of rewrite rules in lindenmayer systems and state transition rules in cellular automata via genetic programming. In *Symposium on Pattern Formation (SPF-93), Claremont, California*, 1993.

[11] Daniel Leitner, Sabine Klepsch, Gernot Bodner, and Andrea Schnepf. A dynamic root system growth model based on l-systems. *Plant and Soil*, 332(1-2):177–192, 2010.

[12] Kishore Mulchandani. *Morphological modeling of neurons.* PhD thesis, Texas A&M University, 1995.

[13] Craig G. Nevill-Manning and Ian H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *arXiv preprint cs/9709102*, 1997.

[14] P. et all Prusinkiewicz. Algorithmic botany group. `http://algorithmicbotany.org/`. [Online; accessed 10-May-2013].

[15] Przemyslaw Prusinkiewicz and Brendan Lane. Modeling morphogenesis in multicellular structures with cell complexes and l-systems. In Vincenzo Capasso, Misha Gromov, Annick Harel-Bellan, Nadya Morozova, and Linda Louise Pritchard, editors, *Pattern Formation in Morphogenesis*, volume 15 of *Springer Proceedings in Mathematics*, pages 137–151. Springer Berlin Heidelberg, 2013.

[16] Przemyslaw Prusinkiewicz, Aristid Lindenmayer, James S. Hanan, F. David Fracchia, Deborah R. Fowler, Martin J.M. de Boer, and Lynn Mercer. *The algorithmic beauty of plants*. Springer-Verlag New York, 1990.

[17] Kevin Roast. L-systems turtle graphics renderer. `http://www.kevs3d.co.uk/dev/lsystems/`. [Online; accessed 10-May-2013].

[18] E. Santos and R.C. Coelho. Obtaining l-systems rules from strings. In *Computational Modeling (MCSUL), 2009 Third Southern Conference on*, pages 143–149, 2009.

[19] R. Sebastian, V. Zimmerman, D. Romero, and A.F. Frangi. Construction of a computational anatomical model of the peripheral cardiac conduction system. *Biomedical Engineering, IEEE Transactions on*, 58(12):3479–3482, 2011.

[20] Rafael Sebastian, Viviana Zimmerman, Daniel Romero, Damian Sanchez-Quintana, and A Frangi. Characterization and modeling of the peripheral cardiac conduction system. 2013.

[21] Ekaterina H. Taralova, Joseph Schlecht, Kobus Barnard, and Barry M. Pryor. Modelling and visualizing morphology in the fungus alternaria. *Fungal Biology*, 115(11):1163 – 1173, 2011.

[22] Mair Zamir. Arterial branching within the confines of fractal l-system formalism. *The Journal of general physiology*, 118(3):267–276, 2001.

Appendix A: Python code

```
'''
Code by Judith van Stegeren
June 3, 2013
v3

This code was written to illustrate and validate the
algorithm for inference of L-systems from simple strings.
The original algorithm together with an example and
pseudo-code can be found in the paper "Obtaining L-system
rules from strings" by Santos and Coelho (2012).

This algorithm was tested with the following strings:

    s = "F+F-+F+F--"
    found rules:
        F-> F+F-

    s = "F+F+++F+F++++"
    found rules:
        F-> F+F++

    s = "F+[F+F]-+[F+[F+F]-+F+[F+F]-]-"
    found rules:
        F-> F+[F+F]-
'''

import math
import re

# counts the number of non-terminals in string s
def countF(s):
    n = 0
    for char in s:
        if char == 'F':
            n+=1
    return n

#generate possible tuples for Fr, n, such that Fs = Fr ** n
def bftuples(Fs):
    tuples = []
    for Fr in range(2,int(math.ceil(math.sqrt(Fs)))+1):
        n = (math.log(Fs)/math.log(Fr))
        if not n == 0 and n % 1 == 0 and Fs == (Fr ** n):
            tuples.append((Fr,n))
    return tuples

#checks whether pattern occurs exactly k times within s
```

```
48 def checkpatt(s,k,pattern):
49     if len(re.findall(r'%s' %re.escape(pattern),s)) == k:
50         return True
51     return False
52 
53 # Slices s in Fr slices with k times an 'F'
54 # and checks if each slice contains the same pattern
55 def Slice(s,Fs,Fr):
56     k = Fs / Fr
57     cF = 0
58     pattern = ""
59     for char in s:
60         if cF == k:
61             return pattern
62         if char == 'F':
63             pattern+=char
64             cF+=1
65         if not char == 'F' and cF>0:
66             pattern+=char
67     if cF == k and checkpatt(s,k,pattern):
68         return pattern
69     return ""
70 
71 # replace all occurences of oldpatt with replacement
72 def Replace(string,oldpatt,replacement):
73     if oldpatt == replacement:
74         return string
75     return re.sub('%s' % re.escape(oldpatt),replacement,string)
76 
77 # calculates the number of leading and trailing terminal
78 # characters
79 def findleading(string):
80     leading = len(string)-len(string.lstrip("+-[]"))
81     trailing = len(string)-len(string.rstrip("+-[]"))
82     return (leading,trailing)
83 
84 
85 # strips the leading and trailing from each substring
86 # with 'F' in string
87 def strip(string,i,left):
88     new_string = ""
89     if not left:
90         for substring in re.findall(r'F[\-\+\]\[]+',string):
91             counter = i
92             new_substring = ""
93             for char in substring:
94                 if char == 'F' or counter == 0:
95                     new_substring += char
96                 else:
97                     counter-=1 #delete a character
```

```
98              new_string += new_substring
99      else:
100         for substring in re.findall('[\[\]\-\+]+F',string):
101             counter = i
102             new_substring = ""
103             for char in substring[::-1]:
104                 if char == 'F' or counter == 0:
105                     new_substring += char
106                 else:
107                     counter-=1
108             new_string += new_substring[::-1]
109     return new_string
110 
111 #cleans up the accumulated terminal characters in string
112 def TerminalClean(string,n):
113     (l,r) = findleading(string)
114     l = (l / n) * (n-1)
115     r = (r / n) * (n-1)
116     if l > 0:
117         string = strip(string,l,True)
118     if r > 0:
119         string = strip(string,r,False)
120     return string
121 
122 # returns whether it's possible to make s by applying
123 # rule to axiom and deriving s in exactly n steps
124 def isPossible(axiom,rule,n,s):
125     for i in range(int(n)):
126         axiom = re.sub('%s' % re.escape("F"),rule,axiom)
127     if axiom == s:
128         return True
129     return False
130 
131 # the algorithm to infer D0L-systems from a single string,
132 # as written down in 'Obtaining L-system Rules from Strings'
133 # by Santos & Coelho.
134 # This implementation contains small edits as proposed in
135 # the bachelor thesis of Judith van Stegeren
136 def paperalgo():
137 
138     s = "F+[F+F]-+[F+[F+F]-+F+[F+F]-]-"
139     Fs = countF(s)
140     rules = []
141     rules.append(s)
142     for (Fr,n) in bftuples(Fs):
143         oldpatt = Slice(s,Fs,Fr)
144         string = Replace(s,oldpatt,'F')
145         if countF(string)==Fr:
146             rule_tail = (TerminalClean(string,n))
147             if isPossible('F',rule_tail,n,s):
```

```
148                 rules.append(rule_tail)
149 
150     print "------Rules-------"
151     for rule in rules:
152         print "F->",rule
153 
154     return 1
155 
156 def main():
157     paperalgo()
158 
159 if __name__ == '__main__':
160     main()
```