Calculating the ith Letter of the nth Word in a D0L-sequence

by

David Swart

A thesis

presented to the University of Waterloo

in fulfilment of the

thesis requirement for the degree of

Master of Mathematics

in

Computer Science

Waterloo, Ontario, Canada, 2001

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Let Σ be a finite alphabet and let Σ^* be the finite set of words with letters chosen from Σ. A homomorphism is a map $\varphi : \Sigma^* \to \Sigma^*$, such that for any $x, y \in \Sigma^*$, $\varphi(xy) = \varphi(x)\varphi(y)$. Homomorphisms have applications in formal languages and number theory. As used in D0L-systems, homomorphisms also have applications in computer graphics, and theoretical biology.

We discuss an algorithm which takes a homomorphism $\varphi : \Sigma^* \to \Sigma^*$, integers i and n, and a letter $a \in \Sigma$ as its inputs, and calculates the ith letter of the word $\varphi^n(a)$ in polynomial time. By *polynomial time* we mean polynomial in the size of the input: $\log n$, $\log i$, and the description size of φ. We also discuss a modification which allows us to calculate the letter distribution of subwords of $\varphi^n(a)$.

In the discussion of our new algorithm, we describe non-trivial subroutines for the efficient computation of the lengths of certain word. Let $x \in \Sigma^*$, $a \in \Sigma$, and n,q,r,s, and t be integers such that $t \leq s$, and $r < q$, then we provide subroutines for the calculation of $|\varphi^n(a)|$ and $|\varphi^{tq+r}(x)\ \varphi^{(t+1)q+r}(x) \cdots \varphi^{sq+r}(x)|$.

Further, we see that by increasing the computational power of the homomorphism φ to that of a finite-state transducer, the corresponding calculation becomes exponentially hard.

Acknowledgements

First and foremost, I would like to acknowledge my supervisor Jeffrey Shallit who helped make my academic experience a positive one. He acquainted me with the field of formal languages, encouraged me to pursue my Masters degree, introduced me to my thesis topic and suggested a structure for its solution, gave me valuable pointers on layout, style, and writing technique, and gave me many opportunities to speak about my topic to others.

I wish to show appreciation for all my mathematics and computer science teachers who helped maintain and encourage my interest in such matters (most notably my father Jerry Swart, Mike Elsie, Rick Kitto, Steve Furino and Prabhakar Ragde).

And finally, thanks go to my parents, family, and friends for being part of my life outside of school and for helping me keep my priorities in check. Of these, I would especially like to thank my fiancée, Cindy Crnac, who gave me all sorts of support, even before I decided to enroll in grad school.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Consider taking a given word and simultaneously replacing each letter with some word according to a set of rewriting rules.

For example, consider the following rewriting rules:

$$\begin{aligned} \mathtt{a} &\rightarrow \mathtt{na} \\ \mathtt{b} &\rightarrow \mathtt{ba} \\ \mathtt{n} &\rightarrow \epsilon \end{aligned}$$

where ϵ denotes the empty string of length 0 and $\rightarrow$ denotes rewriting. The simultaneous application of these rules to the word `ban` replaces `b` with `ba`, `a` with `na`, and in essence, deletes the letter `n`, yielding the word `bana`.

Now consider the sequence of words obtained by repeatedly applying these rewriting rules to each successive word. For our example, let us start with `ba` to obtain the infinite sequence:

$$\mathtt{ba}, \mathtt{bana}, \mathtt{banana}, \mathtt{bananana}, \ldots,$$

or, to put it another way [San98]:

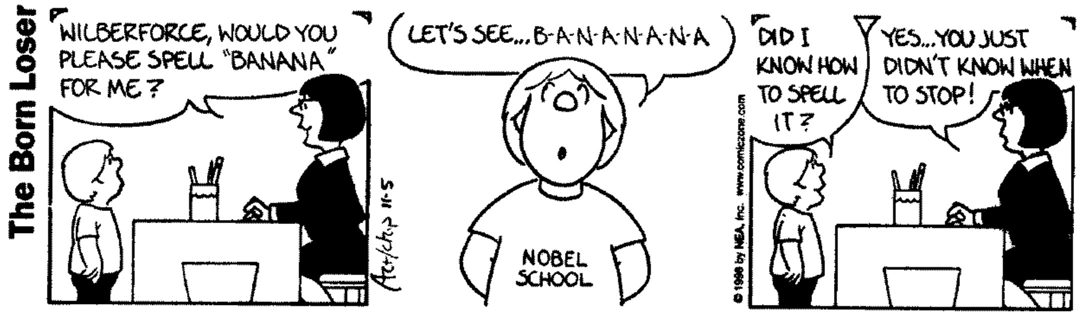

Figure 1.1: The Born Loser. November 5, 1998

These rewriting rules and their associated sequences are well-studied concepts in computer science. In formal languages, the functions which apply these rewriting rules are called homomorphisms. Accompanied by an initial word, homomorphisms and their associated sequences are are sometimes referred to as D0L-systems and D0L-sequences respectively [Vit80].

The D0L-sequence in the above example is fairly simple: each consecutive word is obtained by appending `na` to the previous word. This simplicity may be misleading, since there are D0L-sequences in which the lengths of words grow exponentially and in which the words themselves have a complex structure. For example, starting with `b`, the repeated application of `a` $\rightarrow$ `ab` and `b` $\rightarrow$ `a` yields

`b`, `a`, `ab`, `aba`, `abaab`, `abaababa`, `abaababaabaab`, ...

whose word lengths from the famous Fibonacci sequence.

The time required to calculate the first n words of an arbitrary D0L-sequence is, evidently, at least as long as the time needed to output the answer. If the length of the last word calculated is exponentially large in terms of n, then the calculation takes at least an exponential amount of time as well.

To calculate an individual letter of a specified word, say the nth word, of a D0L-sequence,

$$\left.\begin{array}{rl} \varphi(\mathtt{a}) = & \mathtt{a\ b} \\ \varphi(\mathtt{b}) = & \mathtt{a} \\ \varphi(\mathtt{c}) = & \mathtt{d\ b\ c\ b} \\ \varphi(\mathtt{d}) = & \mathtt{d} \\ \varphi(\mathtt{e}) = & \underbrace{\mathtt{a\ b\ c\ d\ a}}_{\text{width } w} \end{array}\right\} \text{depth } d$$

Figure 1.2: The specification of a homomorphism

we could explicitly write out the sequence until the nth word is encountered and then examine the appropriate letter, but this method is quite inefficient. This thesis presents a method to efficiently calculate such a letter which is efficient, especially for large values of n.

1.1 The ith letter of $\varphi^n(a)$

Before we proceed any further, let us formally define some important concepts and explicitly state the main problem resolved by this thesis.

Let Σ be a finite alphabet and let Σ^* be all the words of finite length that consist of symbols chosen from Σ. Formally, a *homomorphism* is a map φ from Σ^* to Σ^* such that $\varphi(xy) = \varphi(x)\varphi(y)$ for all $x, y \in \Sigma^*$. To describe a homomorphism, we need only define it over Σ. Thus the application of φ to a word in Σ^* entails applying φ to each individual letter and then concatenating the results. We compose a homomorphism φ with itself by defining $\varphi^j(a) = \varphi^{j-1}(\varphi(a))$ for $j > 0$ and $\varphi^0(a) = a$.

Let $|x|$ denote the length of a word $x \in \Sigma^*$. We define the *width* w of a homomorphism φ to be the maximum value of $|\varphi(a)|$ over all $a \in \Sigma$, and the *depth* d of a homomorphism φ to be the cardinality of its domain Σ; for example, see Figure 1.2.

We are now ready to formally define our main problem.

Problem 1 (Main) *Given a homomorphism $\varphi : \Sigma^* \to \Sigma^*$, integers $n \geq 0$ and $i \geq 1$, and a letter $a \in \Sigma$, efficiently calculate the ith letter of $\varphi^n(a)$.*

The size of the input data for any algorithm that solves this problem is $O(wd + (\log n) + (\log i))$.

Our model of computation is the familiar "naive bit complexity" model; see, for example, [BS96]. In this model, the addition of two n-bit integers uses $O(n)$ bit operations, while the multiplication of two n-bit integers uses $O(n^2)$ bit operations.

The brute force solution to Problem 1 calculates each word a, $\varphi(a)$, $\varphi^2(a)$, …, $\varphi^n(a)$, and then returns the ith letter of the final word. In the worst case, the running time of the brute force method is $\Omega(w^n)$ which is doubly exponential in the size of the input data.

We present the first algorithm which solves Problem 1 in polynomial time in the size of the input. More precisely, the running time of our algorithm is polynomial in $\log n$, $\log i$, w, and d.

1.2 Overview

In Chapter 2 we discuss various aspects of homomorphisms and D0L-systems such as D0L-languages, the growth function of a D0L-system, other L-systems, and fixed points of homomorphisms. We also demonstrate some of the uses of our algorithm.

Chapter 3 provides descriptions and analyses of the subroutines used by our main algorithm. GenericLength and Length are used to calculate the lengths of specific forms of words and the subroutine BasicTreeDescent is used as a base case.

In Chapter 4, we present the main algorithm for Find and provide some sample executions. We end the chapter with an analysis of the algorithm's correctness and theoretical running time as well as empirical test results of the actual running time.

Our algorithm has been implemented and tested in C++. Chapter 5 provides the details of this implementation.

Let $|x|_a$ denote the number of occurrences of the letter a in x, for $a \in \Sigma$ and $x \in \Sigma^*$, The *Parikh vector* of a word x is the vector of non-negative integers describing the letter distribution of x, that is, $(m_a)_{a\in\Sigma}$, where $m_a = |x|_a$. In Chapter 6, we show how to extend our algorithm to return the Parikh vector of the prefix of $\varphi^n(a)$ of length i. Chapter 6 also considers two other classes of functions with more computational power than homomorphisms. We demonstrate that the computation of the ith letter of words produced with such functions requires at least an exponential amount of time. We end Chapter 6 by mentioning some related open problems.

Chapter 2

Previous Work

2.1 D0L-systems

Homomorphisms are most often studied in the form of of Lindenmayer systems or, L-systems, which were developed to model the development of biological organisms; see, for example, Lindenmayer and Jürgensen [LJ92] for a survey of ideas and results focusing mainly on the application of L-systems to theoretical biology.

In L-system terminology, a homomorphism together with an initial word is called a *deterministic L-system without interactions*, since homomorphisms replace letters by a specific word depending only upon the replaced letter and without any interaction from neighboring letters. If we let "D" stand for *deterministic* and "0" stand for *without interactions* then we have the term *D0L-system*. Formally, a D0L-system is a 3-tuple $G = (\Sigma, \varphi, z)$ where Σ is a finite alphabet, φ is a homomorphism which acts on Σ^*, and $z \in \Sigma^*$ is the initial word or *axiom*.

In their simplest interpretations, words produced by D0L-systems represent a linear string of cells, with different symbols representing various properties of each cell. Szilard and Quinton [SQ79] suggested a new way to interpret these words to model two and three dimensional structures. A *turtle language* is any programming language such as Logo [Fri88] that has a set of commands to a "turtle" to move around a plane and leave trail. By interpreting the characters of a word of a D0L-sequence as the instructions for a turtle language, we can produce images such as the branching structure of a tree or finite approximations of fractals.

Many papers have focused on the use of L-systems to generate images and structures. Prusinkiewicz and Hanan [PH89] and Demko, Hodges, and Naylor [DHN85] provided many examples of plant images and fractals which can be produced by L-systems. Prusinkiewicz and Lindenmayer [PL90] explored the ability of L-systems to model aspects of plant growth such as branching structures, and leaves. While rendering a maple tree, Bloomenthal [Blo85] used L-systems to model its branching structure.

Prusinkiewicz and Lindenmayer [PL90] provided the following examples to illustrate how D0L-systems create fractals and trees.

Consider the D0L-system $G_1 = (\Sigma_1, \varphi_1, z_1)$ where

$$\begin{aligned} \Sigma_1 = \;& \{\mathtt{f}_l, \mathtt{f}_r, \mathtt{r}, \mathtt{l}\} \\ \varphi_1 = \;& \mathtt{f}_l \rightarrow \mathtt{f}_l\mathtt{l}\mathtt{f}_r \\ & \mathtt{f}_r \rightarrow \mathtt{f}_l\mathtt{r}\mathtt{f}_r \\ & \mathtt{l} \rightarrow \mathtt{l} \\ & \mathtt{r} \rightarrow \mathtt{r}, \\ z_1 = \;& \mathtt{f}_l. \end{aligned}$$

Then the G_1 produces the following sequence of words,

$$\mathtt{f}_l,\ \mathtt{f}_l\mathtt{l}\mathtt{f}_r,\ \mathtt{f}_l\mathtt{l}\mathtt{f}_r\mathtt{l}\mathtt{f}_l\mathtt{r}\mathtt{f}_r,\ \mathtt{f}_l\mathtt{l}\mathtt{f}_r\mathtt{l}\mathtt{f}_l\mathtt{r}\mathtt{f}_r\mathtt{l}\mathtt{f}_l\mathtt{l}\mathtt{f}_r\mathtt{r}\mathtt{f}_l\mathtt{r}\mathtt{f}_r,\ \ldots$$

If we interpret the f's as an instruction to move forward, and r, and l as instructions to make 90° right and left turns respectively, then the eleventh word of this sequence produces the fractal in Figure 2.1, known as a Heighway Dragon [Gar67].

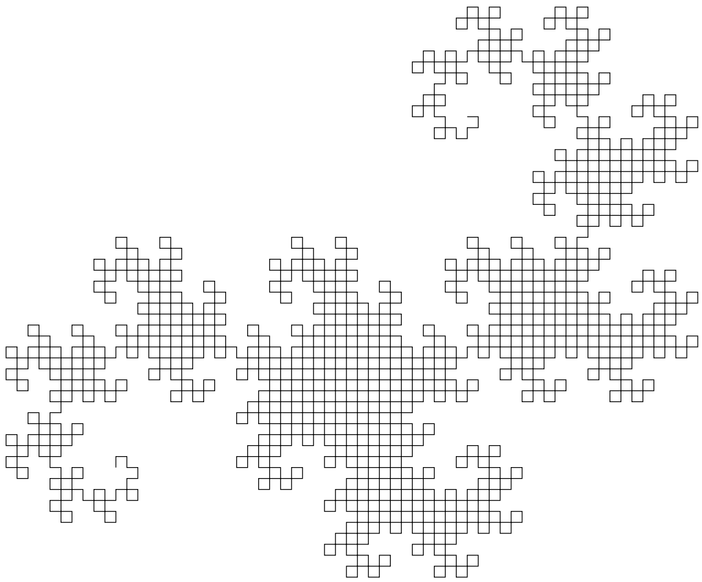

Figure 2.1: The Heighway Dragon.

Now consider the D0L-system $G_2 = (\Sigma_2, \varphi_2, z_2)$, where

$$\begin{aligned}
\Sigma_2 = \ & \{\mathtt{x}, \mathtt{f}, \mathtt{r}, \mathtt{l}, \mathtt{[}, \mathtt{]}\}, \\
\varphi_2 = \ & \mathtt{x} \rightarrow \mathtt{fl[[x]rx]rf[rfx]lx} \\
& \mathtt{f} \rightarrow \mathtt{ff} \\
& \mathtt{r} \rightarrow \mathtt{r} \\
& \mathtt{l} \rightarrow \mathtt{l} \\
& \mathtt{[} \rightarrow \mathtt{[} \\
& \mathtt{]} \rightarrow \mathtt{]}, \\
z_2 = \ & \mathtt{x}.
\end{aligned}$$

Here, the right and left turns are only 22.5°, the symbol `x` has no effect on the turtle, and brackets are used to delimit branches, that is, when a `]` is encountered, the position and direction of the turtle are restored to when the matching `[` was encountered.

Figure 2.2 shows the graphical interpretation of the fifth word of the sequence produced by G_2.

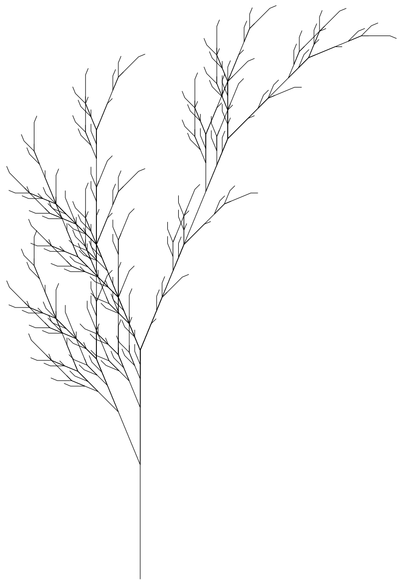

Figure 2.2: A tree produced by the D0L-system G_2.

While much attention has been paid to the applications of D0L-systems, a significant amount of study has also been devoted to their theoretical properties; see, for example, Paul Vitányi's thesis [Vit80] for a comprehensive collection of theoretical results up to 1980.

We want to know, for example, about the modeling capabilities of D0L-systems, that is, which sequences of words can D0L-systems produce, and what kinds of growth can D0L-systems imitate? The remainder of this chapter deals mainly with such theoretical questions.

2.1.1 Locally Catenative D0L-systems

Often, pictures produced by D0L-systems such as the fractal and the branch in Figures 2.1 and 2.2 have a self-similar quality, that is, we can see earlier or younger versions of the image within the finer details of the image itself. The term *locally catenative* applies to those D0L-sequences with words comprised *entirely* of earlier words in the sequence. Recall that we use z for the axiom and φ for the homomorphism of a D0L-system. Formally, a D0L-system is locally catenative if and only if

$$\varphi^n(z) = \varphi^{n-i_1}(z)\varphi^{n-i_2}(z)\cdots\varphi^{n-i_k}(z)$$

for all n greater than or equal to some value n_0 and for integers $i_1, \ldots, i_k \leq n_0$. The *depth* of a locally catenative system is the maximum of $i_1, \ldots, i_k$.

Whether a given D0L-sequence is locally catenative or not is a long-standing open problem in L-systems. The problem was posed originally by Lindenmayer and Rozenberg [LR73] and in the intervening 25 years, there have been partial results on this problem.

Linna [Lin77] showed the decidability of the D0L prefix problem: does there exist some word x_1 in a given D0L-sequence which is a proper prefix of a word x_2 occurring later in the same sequence? Ehrenfeucht and Rozenberg [ER78] showed we can decide whether a system is locally catenative with a depth less than a given value, but decidability for arbitrary depth still remains to be shown. Kobuchi and Seki [KS79] showed that the analogous problem corresponding to DIL-systems — an interactive version of D0L-systems explained in Section 2.2 — is *not* decidable, even for the barely interactive D1L-systems. More recently, Choffrut [Cho90, Cho92] demonstrated that the property of being locally catenative *is* decidable for binary D0L-systems, that is, systems with an alphabet of two letters.

2.1.2 Tools for Discussing D0L-systems

The behavior of a D0L-sequence depends largely upon the way letters in Σ recursively produce themselves. It makes sense then, to classify letters of a D0L-system in terms of their recursive structure. First, some formal definitions are required.

Recall that ϵ denotes the empty string. A letter a is *mortal* if $\varphi^n(a) = \epsilon$ for some $n > 0$. We say a is *immortal* or *vital* if it is not mortal. A letter b is *accessible* from a if there exist $n \geq 0$ and $x_1, x_2 \in \Sigma^*$ such that $\varphi^n(a) = x_1 b x_2$. We say a is *recursive* if a is accessible from itself. We say a is *mono-recursive* if $\varphi^n(a) = x_1 a x_2$ for some integer $n > 0$ and $x_1, x_2 \in \Sigma^*$ such that a is not accessible from x_1x_2. Finally, a letter a is *expanding* when $\varphi^n(a) = x_1ax_2ax_3$ for some integer $n > 0$ and some $x_1, x_2, x_3 \in \Sigma^*$. As we see below in Section 2.4, a recursive letter is either mono-recursive or expanding. Vitányi [Vit80] provided efficient algorithms to determine which letters are immortal, mortal, recursive, mono-recursive, expanding and accessible from some given letter.

Vitányi [Vit76, Vit80] classified the recursive structure of a D0L-system with the definitions of specific directed graphs. The *associated digraph* of a D0L-system G, denoted by $\mathrm{AD}(G)$, is a directed graph with vertices belonging to Σ and arcs from a to b if and only if $\varphi(a)$ contains the letter b (loops are allowed). The *condensed associated digraph* or $\mathrm{CAD}(G)$ is obtained from $\mathrm{AD}(G)$ by contracting each cycle into a vertex. Formally, the vertices of $\mathrm{CAD}(G)$ are labeled with the sets of mutually accessible letters of Σ. and there is an arc from a vertex $\{a_1, a_2, \ldots, a_m\}$ to a vertex $\{b_1, b_2, \ldots, b_n\}$ if and only if there exist a_i and b_j such that $\varphi(a_i)$ contains the letter b_j. We obtain the *recursive structure* of G or $\mathrm{RS}(G)$ from $\mathrm{CAD}(G)$ by deleting those vertices labeled $\{a\}$ where a is not a recursive letter. Finally, the *unlabeled recursive structure* of G or $\mathrm{URS}(G)$ is obtained by simply removing the labels of $\mathrm{RS}(G)$. In Section 2.3.2, we will see some results Vitányi obtained by using these digraphs.

While $\mathrm{AD}(G)$, $\mathrm{CAD}(G)$, $\mathrm{RS}(G)$, and $\mathrm{URS}(G)$ allow us to describe the recursive structure of a D0L-system, we use another structure to examine how individual words of a D0L-system are derived.

Let the axiom z equal a letter $a \in \Sigma$. We view the first $n+1$ words of the D0L-sequence a, $\varphi(a)$, $\varphi^2(a)$, $\ldots$, $\varphi^n(a)$ as labels of the levels of an ordered tree. More precisely, the *derivation tree* of $\varphi^n(a)$ is the ordered tree of height n, with a as its root, such that every non-leaf node b has children labeled with the letters of $\varphi(b)$.

A *derivation path* is a path which leads from the root of a derivation tree to some node of the tree. Derivation paths serve a useful role in the analysis of homomorphisms; see, for example, [LR74]. Figure 2.3 shows the derivation tree for $\varphi^5(\mathtt{b})$, where $\varphi(\mathtt{a}) = \mathtt{ab}$ and

$\varphi(\mathtt{b}) = \mathtt{a}$. The derivation path of the last occurrence of b in $\varphi^5(\mathtt{b})$ is indicated by a double line.

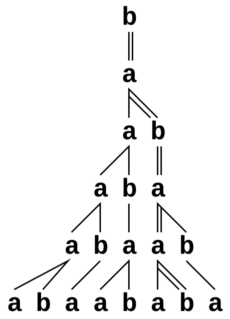

Figure 2.3: The derivation tree of $\varphi^5(\mathtt{b})$. [PL90]

2.2 Other L-systems

D0L-systems are only one kind of Lindenmayer system. We focus on D0L-systems particularly because they provide a means to discuss the sequence of words $a, \varphi(a), \varphi^2(a), \ldots$. Other Lindenmayer systems, by comparison, are a departure from modeling the behavior of simple replacement rules. Although the results of this thesis deal mainly with homomorphisms, it is worth examining these other systems to see the context in which D0L-systems are developed. Further, any results which apply to more generalized L-systems are useful since they also apply to D0L-systems themselves.

0L-systems are obtained by introducing nondeterminism to D0L-systems. An 0L-system, for example, may implement the replacement rules $\mathtt{a} \to \mathtt{aaa}$ as well as $\mathtt{a} \to \mathtt{a}$. Then by making appropriate replacement rule choices, the initial word a produces $\mathtt{a}^k$, for any odd k.

A tabled D0L-system or *DT0L-system* is a more restricted form of a nondeterministic Lindenmayer system. Instead of one homomorphism, there is a finite set Φ of homomorphisms. Given an initial word z, a DT0L-system produces the words $\varphi_1(\varphi_2(\cdots(\varphi_k(z))\cdots))$ for $k \geq 0$ and $\varphi_i \in \Phi$. Similar to D0L-systems, DT0L-systems replace each letter simultaneously, but now there is a nondeterministic choice of which homomorphism to apply.

If we combine the tables of DT0L-systems with the nondeterminism of 0L-systems we obtain *T0L-systems* which use tables of nondeterministic homomorphisms.

Extended D0L-systems or *ED0L-systems* are obtained by including a set $V \subseteq \Sigma$ of terminal symbols in their specifications. The set of words that a D0L-system extended by an alphabet V can produce is equal to $\{\varphi^n(z) : n \geq 0\} \cap V^*$. This extension increases the specification capabilities of the D0L-system by allowing a word x to be produced while omitting the words needed to derive x that contain non-terminals. By combining various methods to augment a D0L-system, we obtain *EDT0L-systems*, *E0L-systems*, and *ET0L-systems*.

Recall that a letter $a \in \Sigma$ is mortal if $\varphi^n(a) = \epsilon$ for some integer $n \geq 0$. Propagating D0L-systems or *PD0L-systems* are obtained by requiring that no letter be mortal. As before, systems such as *EPDT0L-systems* and *EP0L-systems* are obtained by augmenting a D0L-system appropriately.

Recall the "0" in D0L signifies that the replacement rules depend only upon the letter being replaced. If we let the replacement rule depend upon k neighbors to the left of the letter being replaced and ℓ neighbors to the right, then we have a D(k, ℓ)L-system which is context-sensitive when k or ℓ is non-zero.

The notation $w\langle a\rangle x \rightarrow y$ where $|w| \leq k$ and $|x| \leq \ell$ indicates that a is replaced by y when preceded by word w and followed by word x, The replacement rules for letters with positions less than k are specified when $|w|$ is strictly less than k, and similarly for letters towards the end of the word and x.

For example, consider the following replacement rules of a D(1,0)L-system:

$$\begin{aligned}
\epsilon\langle \mathsf{a} \rangle &\to \mathsf{b} \\
\mathsf{a}\langle \mathsf{a} \rangle &\to \mathsf{b} \\
\mathsf{b}\langle \mathsf{a} \rangle &\to \mathsf{b} \\
\epsilon\langle \mathsf{b} \rangle &\to \mathsf{b} \\
\mathsf{a}\langle \mathsf{b} \rangle &\to \mathsf{a} \\
\mathsf{b}\langle \mathsf{b} \rangle &\to \mathsf{b}.
\end{aligned}$$

Consider applying these rules simultaneously to all the letters of abbbb. We see that the a is replaced with a b, the b preceded by the a is replaced with an a, and since each b in the remainder of the word is preceded by another b, it is replaced by itself. Hence we obtain the word babbb and by iteration, we obtain the following sequence, beginning with abbbb:

$$\mathsf{abbbb},\ \mathsf{babbb},\ \mathsf{bbabb},\ \mathsf{bbbab},\ \mathsf{bbbba},\ \mathsf{bbbbb},\ \mathsf{bbbbb},\ \mathsf{bbbbb},\ \ldots.$$

The three most commonly discussed D(k, ℓ)L-systems have their own names: we already know of D(0,0)L-systems as D0L-systems, the term *D1L-system* is used for D(0,1)L and D(1,0)L-systems, and D(1,1)L-systems are known as *D2L-systems*. In an unfortunate resemblance to D1L-systems, the term *DIL-system* is used for all the interactive Lindenmayer systems. More precisely, DIL-systems are those D(k, ℓ)L-systems with $k + \ell > 0$.

It should be noted that if we set $V = \Sigma$ then ED0L-systems become simple D0L-systems. By limiting the nondeterministic choices to one choice, 0L-systems and DT0L-systems also specify D0L-systems. And by specifying the same replacement rules for every context, DIL-systems effectively become D0L-systems. Hence, generalized statements about ED0L-systems, 0L-systems, DT0L-systems, DIL-systems, and any of their combinations also apply to D0L-systems. On the other hand the same cannot be said for propagating systems. Rozenberg [Roz74] showed that PD0L-systems are not capable of generating the same sequences or even the same languages that D0L-systems can.

2.3 D0L-languages

The *language* of a D0L-system $G = \{\Sigma, \varphi, z\}$ is defined as $L(G) = \{\varphi^n(z) \ : \ n \geq 0\}$. There has been a great deal of study concerning the class of D0L-languages.

In the field of formal languages, we often want to compare a new class of languages to the well-known Chomsky hierarchy. Recall that the class of D0L-languages are properly contained within the class of 0L-languages. Doucet and Rozenberg [DR71] showed the class of 0L-languages is contained in the class of context-sensitive languages. Therefore, every D0L-language is in CSL. In Theorem 3.44 of [Vit80], Vitányi showed that neither the class of D0L-languages nor the class of regular languages are contained in the other but neither are they disjoint. A similar result holds for context-free languages.

To illustrate that there are D0L-languages which are regular ($L_1 = \{\mathtt{a}\}$), context-free but not regular ($L_2 = \{\mathtt{a}^n\mathtt{b}\mathtt{c}^n; n \geq 0\}$), and context-sensitive but not context-free ($L_3 = \{\mathtt{a}^{2^n}; n \geq 0\}$), Vitányi provided the following D0L-systems.

$$\begin{aligned} G_1 = & \quad (\{\mathtt{a}\}, \{\mathtt{a} \rightarrow \mathtt{a}\}, \mathtt{a}) \\ G_2 = & \quad (\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, \{\mathtt{a} \rightarrow \mathtt{a}, \mathtt{b} \rightarrow \mathtt{b}, \mathtt{c} \rightarrow \mathtt{acb}\}, \mathtt{c}) \\ G_3 = & \quad (\{\mathtt{a}\}, \{\mathtt{a} \rightarrow \mathtt{aa}\}, \mathtt{a}) \end{aligned}$$

See Figure 2.4 for an illustration of D0L-languages in relation to the Chomsky hierarchy.

2.3.1 D0L Membership

D0L-systems are useful for the specification of sequences of words generated by a homomorphism, and given a D0L-system, we can easily generate these sequences. However, once we define *languages* in terms of D0L-systems, we should deal with the decision of whether a given word x belongs to the language $L(G)$ for a given D0L-system G. We call this the *D0L*

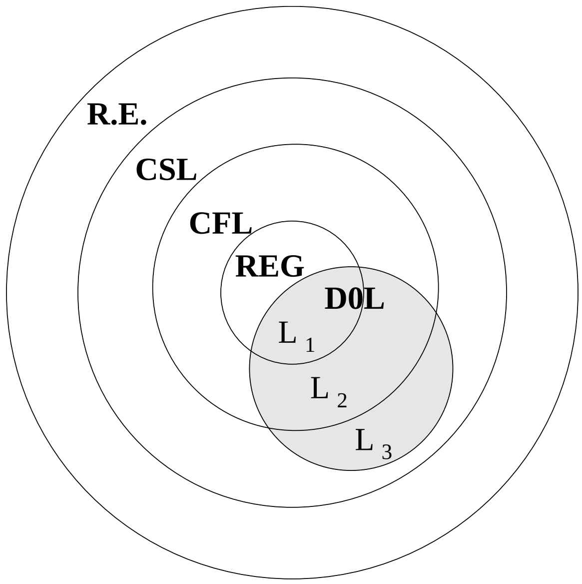

Figure 2.4: The class of D0L-languages and the Chomsky hierarchy.

membership problem.

As pointed out by Vitányi [Vit72], since D0L-languages are in CSL, the membership problem is decidable. Doucet [Dou72] however, showed decidability by giving an explicit, albeit infeasible, algorithm to solve the D0L membership. Vitányi [Vit72, Vit74] proceeded to give a feasible algorithm which "delivers answers (to reasonable questions) in a matter of seconds."

Jones and Skyum [JS77a, JS77c, JS79] established that EDT0L membership is in non-deterministic logarithmic space, or NSPACE($\log n$), which is known to be in P. Therefore, since all D0L-languages are EDT0L-languages, D0L membership is in P. Later, Jones and Skyum [JS81] addressed the D0L membership problem directly and showed the problem to be in DSPACE($\log n$). Thus they concluded the membership problem is not likely to be P-complete.

The slightly different problem of *recognizing* a D0L-language is distinguished from the D0L membership problem by treating $L(G)$ as fixed and having the input consist only of the word x.

Jones and Skyum [JS76, JS77b] first gave a polynomial algorithm to recognize an EDT0L (hence D0L) language. Independently, the same result was obtained by Harju [Har77], and Sudborough [Sud77]. However, Sudborough went further and proved that the more specific case of recognizing an ED0L (and hence D0L) language can be solved in DSPACE($\log n$).

2.3.2 D0L Equivalence

In 1973, Doucet [Dou73] stated two of the most fascinating, yet elusive problems in regard to D0L-languages: what is the decidability of the D0L-language equivalence problem and the D0L-sequence equivalence problem? Consider two D0L-systems $G_1 = (\Sigma, \varphi_1, z_1)$ and $G_2 = (\Sigma, \varphi_2, z_2)$. The *D0L-language equivalence* problem (or just *D0L equivalence* problem) decides whether $L(G_1) = L(G_2)$. The *D0L-sequence equivalence* problem decides whether $\varphi_1^n(z_1) = \varphi_2^n(z_2)$ for $n \geq 0$.

Before decidability was finally solved, several papers were published which yielded partial results. As we see below in Section 2.4, a D0L-system *has rank* if the word lengths of the D0L-system are bounded by a polynomial. Ehrenfeucht and Rozenberg [ER74] showed that the equivalence between two given PD0L-languages with rank is decidable. However, their proof did not apply to D0L-systems without rank or non-propagating D0L-systems.

Salomaa [Sal75a] showed the decidability for the equivalence between a CFL and a D0L-language.

By using his associated digraphs for D0L-systems, Vitányi [Vit76] showed that equivalence for languages of locally catenative D0L-systems is decidable. Vitányi also used associated digraphs to reproduce Ehrenfeucht and Rozenberg's result that the equivalence problem is decidable for propagating D0L-systems with rank.

Finally, Culik and Fris [CF77] showed that both the D0L-sequence equivalence problem and the D0L-language equivalence problem were decidable. Later, Culik [Cul78] showed the decidability of the *ultimate* D0L-sequence equivalence problem: Given two D0L-systems $G_1 = (\Sigma, \varphi_1, z_1)$ and $G_2 = (\Sigma, \varphi_2, z_2)$, and some integer k, does $\varphi_1^n(z_1) = \varphi_2^n(z_2)$ for all

$n \geq k$?

2.3.3 Miscellaneous Results on D0L-languages

Results on D0L-languages are not limited to D0L membership and D0L equivalence.

The *subword complexity of a language* which we denote by $\pi_k(L(G))$, is the number of unique subwords of length k in all the words of $L(G)$. Ehrenfeucht and Rozenberg [ER73] showed that the subword complexity of a 0L-language cannot get very complex. Specifically, their main result states that if G is an 0L-system,

$$\lim_{k\to\infty} \frac{\pi_k(L(G))}{|\Sigma|^k} = 0.$$

Later, Ehrenfeucht, Lee, and Rozenberg [ELR75] showed that when determinism is introduced, the subword complexity is $O(k^2)$. More precisely, if G is a D0L-system, then $\pi_k(L(G)) = O(k^2)$.

Let M, V, R, and MR represent the set of mortal, immortal, recursive and mono-recursive letters of an alphabet Σ respectively. Vitányi [Vit74] gave a criterion to determine whether the language of a D0L-system is finite or not, that is, $L(G)$ is finite if and only if $\varphi^{|V-R|}(z) \in (M \cup MR)^*$. Then, Vitányi gave a function to calculate the maximum sized finite D0L-language given an alphabet of size m and conversely, a function which calculates the minimum size of an alphabet for a D0L-language of size n.

Lee and Rozenberg [LR74] showed that the number of words in a D0L-language of length k are bounded by a constant C_Σ, which depends only upon the size of the alphabet and not upon k, that is, $|\{x \in L(G) : |x| = k\}| \leq C_\Sigma$.

Rozenberg [Roz74] showed that prefixes of infinite D0L-sequences are ultimately periodic. More precisely, let $x_1, x_2, \ldots$ be an infinite D0L-sequence. Then there exists an integer $k \geq 1$ such that the sequence formed by taking the first k letters of $x_1, x_2, \ldots$ is periodic for all but a finite number of prefixes at the start. A similar result holds for suffixes.

2.4 Growth Functions

An important question when we deal with D0L-sequences is the length of their words. The *growth function* f_G of a D0L-system $G = (\Sigma, \varphi, z)$ is defined by $f_G(n) = |\varphi^n(z)|$. By studying growth functions we determine which types of biological growth D0L-systems are capable of modeling. Herman and Vitányi [HV76], for example, provide a summary of some results on growth functions.

Here we discuss a special matrix representation of homomorphisms Paz and Salomaa [PS73] introduced to help calculate the growth function $f_G(n)$.

2.4.1 The Incidence Matrix

The *incidence matrix* of a homomorphism φ is defined as the following square matrix of order d,

$$M(\varphi) = (m_{ab})_{a,b\in\Sigma}, \text{ where } m_{ab} = |\varphi(b)|_a.$$

For example, Figure 2.5 shows the incidence matrix $M(\varphi)$ for the homomorphism which maps $\mathtt{a} \to \mathtt{ab}$, $\mathtt{b} \to \mathtt{a}$, $\mathtt{c} \to \mathtt{dbcb}$, $\mathtt{d} \to \mathtt{d}$, and $\mathtt{e} \to \mathtt{abcda}$.

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 2 \\ 1 & 0 & 2 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 2.5: The incidence matrix of φ

Recall that the Parikh vector of a word x, describes the letter distribution of x. We use incidence matrices by applying them to Parikh vectors. Specifically, if $\mathbf{x}$ is the Parikh vector of $x \in \Sigma^*$, then $M(\varphi)\mathbf{x}$ equals the Parikh vector of $\varphi(x)$. To calculate the Parikh vector for words of the form $\varphi^n(x)$, we need to calculate $M(\varphi^k)$ and multiply it by $\mathbf{x}$. The following lemma demonstrates that by raising $M(\varphi)$ to the kth power, we obtain $M(\varphi^k)$.

Lemma 2 *Let $\varphi : \Sigma^* \to \Sigma^*$ be a homomorphism and let k be an integer ≥ 0, then*

$$M(\varphi)^k = M(\varphi^k).$$

Proof. We proceed by induction on k.

When $k = 0$, $M(\varphi)^k = M(\varphi)^0 = \mathbf{I}$, the identity matrix. Also, $\varphi^k = \varphi^0$, the function which maps each letter to itself. Hence, $M(\varphi^0) = \mathbf{I} = M(\varphi)^0$.

Now assume the statement holds for values less than k. Thus,

$$\begin{aligned} M(\varphi)^k &= M(\varphi)^{k-1}M(\varphi) \\ &= M(\varphi^{k-1})M(\varphi). \end{aligned}$$

Let $a, b \in \Sigma$. The entry at row a and column b of $M(\varphi)^k$ is

$$m_{ab} = \sum_{c \in \Sigma} |\varphi^{k-1}(c)|_a \cdot |\varphi(b)|_c.$$

What this statement says is, m_{ab} is obtained by adding the number of c's in $\varphi(b)$ times the number of a's in $\varphi^{k-1}(c)$ for each $c \in \Sigma$. Hence m_{ab} equals the number of a's in $\varphi^{k-1}(\varphi(b))$ or $|\varphi^k(b)|_a$. Thus, $M(\varphi)^k = M(\varphi^k)$. ■

We now discuss an important identity regarding the growth function of a D0L-system G. Let $\mathbf{e}$ be the column vector whose entries are all ones and let $\mathbf{z}$ represent the Parikh vector of the axiom z. Salomaa [Sal73, Sal75b] stated that

$$f_G(n) = |\varphi^n(z)| = \mathbf{e}^T M(\varphi^n)\mathbf{z}.$$

We see this result holds by observing that $M(\varphi^n)\mathbf{z}$ is the Parikh vector of $\varphi^n(z)$ and further, that the length of a word is the sum of the values in its Parikh vector.

2.4.2 Classification and Terminology

We wish to classify the growth functions of D0L-systems and determine which functions are possible and which functions are not.

We start by asking how fast can the words of a D0L-sequence grow? Recall that the width w of a homomorphism φ is the maximum length of $\varphi(a)$ for all $a \in \Sigma$. We note that the growth function of a D0L-system is limited by an exponential in the width of the homomorphism, that is, $f_G(n) \leq |z|w^n$.

Vitányi [Vit80] classified growth functions into the following four types. A growth function f_G is *exponential* if $f_G(n) \geq c^n$ for some $c > 1$, *polynomial* if $p(n) \geq f_G(n) \geq q(n)$ for some non-constant polynomials $p(n)$ and $q(n)$, *limited* if $0 < f_G(n) < m$ for some constant m, and *terminating* if $f_G(n) = 0$ for some value of n. Salomaa [Sal73] showed that all growth functions are either bounded above by a polynomial or below by an exponential, and Vitányi [Vit80] gave a proof that all functions are either bounded above by a constant or below by an increasing polynomial. Thus, no growth functions other than exponential, polynomial, limited, or terminating exist.

By using the Cayley-Hamilton theorem on incidence matrices, Doucet [Dou73] and Salomaa [Sal73] devised methods to obtain an explicit formula for the growth function. Therefore, we have an algorithm to classify the growth function of an arbitrary D0L-system.

We finish this section with some notes on terminology. For our purposes, we do not need to distinguish limited or terminal growth. Hence, for simplicity, we call any growth function polynomial if it is bounded above by a polynomial and treat limited and terminal growth as special cases of polynomial growth.

The classification of growth functions also applies to letters in Σ. For example, we say a letter a is *exponential* or *grows exponentially* if $f_G(n)$ is exponential, where G is the D0L-system with its axiom $z = a$.

2.4.3 Exponential and Polynomial Growth

Some of the earliest work on growth functions was done by Szilard [Szi71] who focused mainly on polynomial growth by discussing the class of generating functions $F_G(x) = \sum_{n=0}^{\infty} f_G(n)x^n$. For example, every polynomial of degree k and positive integer coefficients is the growth function of a D0L-system with $\Sigma = \{a_i : 1 \leq i \leq k\} \cup \{b_i : 1 \leq i \leq l\}$, such that $|\varphi(a_i)|_{a_i} \leq 1$, $|\varphi(a_i)|_{a_j} = 0$ for $j > i$, and b_i has limited growth.

Salomaa [Sal73] concentrated on the distinction between polynomial and exponential growth although he used the terms *normal* and *malignant* respectively. This terminology is appropriate in the context of the growth of plants. For example, the growth tissues of many species of cacti will sometimes — for reasons unknown — undergo continual subdivision and form fan-shaped crests, a phenomenon known as *fasciation* [Ben82, pp. 17,22].

Recall that a letter a is expanding when $\varphi^i(a) = x_1ax_2ax_3$ and a letter a is accessible from b when $\varphi^i(b) = x_1ax_2$. Salomaa used Szilard's result on polynomially growing D0L-systems and showed that a growth function f_G is malignant if and only if there is an expanding letter a which is accessible from z.

In the interest of further classifying the types of polynomial growth, Ehrenfeucht and Rozenberg [ER74] described a recursive definition for the *rank* of all polynomially growing letters: Essentially, a letter a has rank 1 if its growth is limited. We assign rank $j+1$ to a letter a, if a has limited growth after deleting the letters of rank $\leq j$ from Σ and from the replacement words of φ. If the growth of a letter is not bounded by a polynomial then its rank is not defined. The rank of a D0L-system G is the largest rank of the letters in its axiom z. A D0L-system has no rank if it grows exponentially

Salomaa [Sal73], Vitányi [Vit73], and Paz and Salomaa [PS73] gave efficient algorithms to determine whether a given letter grows exponentially or polynomially. Later, Vitányi [Vit80] gave the slowest possible growth functions for each type of growth. Recall that d is the depth of the homomorphism φ, that is, $d = |\Sigma|$. Vitányi showed, for example, that any exponentially growing D0L-system G has a growth function $f_G(n) \geq 2^{\lfloor n/d \rfloor}$. The proof relies on the fact that $\varphi^d(a)$ contains at least two exponentially growing letters if and only if a grows exponentially. We will need this result later for the proof of Theorem 16.

2.4.4 Miscellaneous Results on Growth Functions

In a similar problem to the D0L-language equivalence problem, we wish to test the equivalence of the growth functions for two arbitrary D0L-systems. Towards this end, Szilard [Szi71] defined a normal form for a subclass of his generating functions.

Paz and Salomaa [PS73] showed that the growth equivalence problem is solvable by giving a method to specify the generating function for the growth of a D0L-system in the form of $p(x)/q(x)$. If the growth functions of D0L-systems G and G' have generating functions $p(x)/q(x)$ and $p'(x)/q'(x)$ respectively, then G and G' have equivalent growth if and only if $p'(x)q(x) = p(x)q'(x)$.

Doucet [Dou73] gave a partial answer to the *D0L realization* problem: determine whether there exists a D0L-system which satisfies a given growth function. As we have pointed out in Section 2.4.3, Szilard has solved this problem for polynomials with positive coefficients. Paz and Salomaa [PS73] offered a procedure which gives the required D0L-system if one exists; otherwise it runs forever.

2.5 Fixed Points

When modeling biological systems with L-systems, the concept of adult organisms — organisms that do not grow and stay in a stable state — arises naturally. In D0L-systems particularly, an *adult string* (also *stationary word* or *k-stable word*) is a word x such that $\varphi^k(x) = x$ for some integer k. The stationary language of an L-system is the set of stationary words of an L-system. Vitányi [Vit74] showed that any finite D0L-language is also a stationary language.

Other results on stationary languages are concerned with how they relate to the Chomsky hierarchy. For instance, Grefenstette [Gre83] gives a characterization of the Chomsky hierarchy using the stationary languages of various L-systems.

Consider a language obtained by taking the union of stationary languages of an 0L-system over all possible axioms. Herman and Walker [HW75] showed that the class of such languages is properly contained in the class of regular languages. However the class of languages obtained from the union of stationary languages over a finite set of axioms is exactly CFL.

Thierrin [Thi79] stated that a language is context-free if and only if it is the stationary language of a PT0L-system and that all T0L-languages are the image under a coding of the stationary language of a T0L-system.

A special kind of adult string is one which produces itself in one step. A *fixed point* of a homomorphism φ is any word x such that $\varphi(x) = x$.

The *equality language* for a given a set of homomorphisms $\varphi_1, \cdots, \varphi_n$, is defined as $\{x \in \Sigma^* : \varphi_1(x) = \varphi_2(x) = \cdots = \varphi_n\}$. Engelfriet and Rozenberg [ER79] studied fixed points languages as a special case of equality languages between a homomorphism φ and the identity mapping.

Fixed points need not be restricted to Σ^*. Consider the one-sided infinite words over the alphabet Σ

$$\Sigma^\omega = \{a_1 a_2 a_3 \cdots : a_i \in \Sigma \text{ for all } i \geq 1\},$$

and the two-sided infinite words

$$\Sigma^{\mathbb{Z}} = \{\cdots a_{-1} a_0 a_1 \cdots : a_i \in \Sigma \text{ for all } i \in \mathbb{Z}\}.$$

Head and Lando [Hea81, HL86] characterized all the finite and one-sided infinite fixed points of a homomorphism and later, Hamm and Shallit provided a different proof [HS98]. Further, Shallit and Wang [SW98] characterized the two-sided infinite fixed points.

Let a be a letter such that $\varphi(a) = ax$, where x is a word containing at least one immortal letter. Then φ has a unique one-sided infinite fixed point starting with a, of the form

$$\varphi^{\omega}(a) = a\, x\, \varphi(x)\, \varphi^2(x)\, \varphi^3(x) \cdots.$$

Many common sequences are one-sided infinite fixed points of this form and have applications in the field of number theory [AS98].

For example, the famous Fibonacci sequence `abaababaabaabab`$\cdots$ is an infinite fixed point of `a` $\to$ `ab` and `b` $\to$ `a`; the Thue-Morse sequence `011010011001`$\cdots$ is a fixed point of `0` $\to$ `01` and `1` $\to$ `10`; and the fixed point for `b` $\to$ `b1`, `1` $\to$ `10`, `0` $\to$ `00` gives the infinite sequence `b110100010`$\cdots$ which contains `1`'s only at the letters corresponding to powers of 2 (if we start indexing at 0).

2.6 Using our Algorithm

We now concentrate on some of the aspects of homomorphisms in which an efficient algorithm to compute the ith letter of $\varphi^n(a)$ could be useful.

Our algorithm computes the ith letter of $\varphi^n(a)$ where a is a *single symbol* in Σ; however, the nth word of a D0L-sequence equals $\varphi^n(z)$ for some *word* $z \in \Sigma^*$. We are still able to calculate the ith letter of the nth word of a D0L-sequence by adding a new symbol a to Σ such that $\varphi(a) = z$ and then calculating the ith letter of $\varphi^{n+1}(a)$. Note that w, the width of the homomorphism, could potentially be increased to $|z|$.

In the context of modeling biological systems such as branching structures by using D0L-systems, our new algorithm allows us to efficiently calculate the structure of small twigs, without having to calculate the structure of the entire tree.

Another application of our algorithm is the efficient computation of the ith letter of a fixed point of a homomorphism, which was stated as an open problem in [Sha98]. We efficiently compute the ith letter of such a fixed point by using binary search to determine which factor the ith letter lies in, and then using our algorithm to find the appropriate letter within a factor. Such a binary search uses an efficient subroutine for computing $|x\varphi(x)\varphi^2(x)\cdots\varphi^n(x)|$, which is given in the algorithm GENERICLENGTH below in Section 3.2.

2.6.1 A Fixed Point Example

We can demonstrate the usefulness of our algorithm by means of a peculiar result from number theory. The homomorphism that maps $\mathtt{1} \rightarrow \mathtt{121}$ and $\mathtt{2} \rightarrow \mathtt{12221}$ has the infinite fixed point $u = u_0u_1u_2u_3\cdots = \mathtt{12112221}\cdots$. The sequence u has the interesting property that $u_{16i+1} = u_{64i+1}$ for $0 \leq i \leq 1864134$ but not for $i = 1864135$. The example demonstrates the need to compute letters with large indices efficiently since there are such letters with interesting properties.

2.6.2 The Cyclic Tower of Hanoi Problem

The tower of Hanoi is a puzzle with disks of unique sizes that fit onto three pegs which we name 1, 2, and 3. The disks start on peg 1, ordered by size with the largest disk at the bottom. The objective of the puzzle is to move all the disks to peg 2 (or 3) by moving one disk at a time, and never having a larger disk above a smaller one. Figure 2.6 shows a solution involving three disks.

The *cyclic* tower of Hanoi problem restricts the movements of the disks to one direction. The only possible moves now are $1 \rightarrow 2$, $2 \rightarrow 3$, and $3 \rightarrow 1$ provided such moves do not place a larger disk upon a smaller one.

Let $\Sigma = \{\mathtt{f}, \mathtt{g}, \mathtt{h}, \mathtt{u}, \mathtt{v}, \mathtt{w}\}$, and let $\varphi = \{\mathtt{f} \rightarrow \mathtt{fvf}, \mathtt{g} \rightarrow \mathtt{gwg}, \mathtt{h} \rightarrow \mathtt{huh}, \mathtt{u} \rightarrow \mathtt{fg}, \mathtt{v} \rightarrow \mathtt{gh}, \mathtt{w} \rightarrow \mathtt{hf}\}$. Further, let $\mathbf{h} \in \Sigma^*$ be an infinite fixed point of φ starting with $\mathtt{f}$, that is, $\mathbf{h} = \mathtt{fvfghfvfgwghuhfvf}\cdots$. Allouche and Dress [AD90] showed that if we interpret $\mathtt{f}$ and

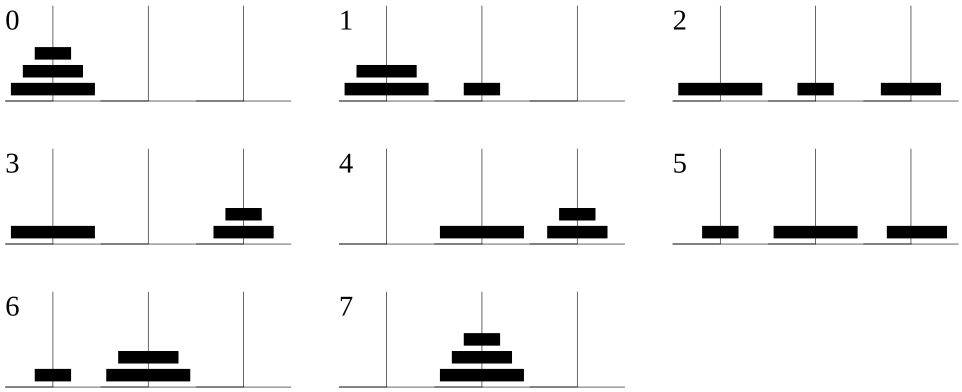

Figure 2.6: Tower of Hanoi.

`w` as $1 \to 2$; `g` and `u` as $2 \to 3$; and `h` and `v` as $3 \to 1$, then a prefix of **h** of the appropriate size yields the optimal solution to the cyclic tower of Hanoi problem.

We can use our algorithm then, to calculate letters of **h**. For example, the 10^{50}th letter of **h** is `f`, an instruction to move a disk from peg 1 to 2. It turns out that the 2^{150}th letter is `w`, also an instruction to move a disk from peg 1 to peg 2.

Chapter 3

Subroutines

Before solving Problem 1, we solve some intermediate problems by describing three subroutines: Length, GenericLength, and BasicTreeDescent. The following is the description and analysis of each subroutine.

3.1 Length

Formally, Length solves the following problem.

Problem 3 *Given a homomorphism $\varphi : \Sigma^* \to \Sigma^*$ of width w, depth d, and given integers n and i and a letter $a \in \Sigma$, calculate*

$$\begin{cases} |\varphi^n(a)|, \text{ if } |\varphi^n(a)| < i; \\ -1, \text{ otherwise,} \end{cases}$$

in time polynomial in $\log n$, $\log i$, *w, and d.*

Since the growth function $f(n) = |\varphi^n(a)|$ can grow exponentially, we know the number of digits in the base-k representation of $|\varphi^n(a)|$ potentially grows linearly with n. Thus $|\varphi^n(a)|$ would be impossible to calculate in time polynomial in $\log n$. We have avoided this problem by observing that, for our purposes, it suffices to report $|\varphi^n(a)| \geq i$ without having to actually calculate $|\varphi^n(a)|$.

Recall that calculating $|\varphi^n(a)|$ is equivalent to calculating $\mathbf{e}^T M(\varphi^n)\mathbf{a}$, where $\mathbf{a}$ is the Parikh vector of a, and $\mathbf{e}$ is the vector containing a 1 for each of its entries. Hence, a straightforward procedure to calculate $|\varphi^n(a)|$, constructs the matrix $M(\varphi)$, multiplies $M(\varphi)$ by itself n times to obtain $M(\varphi^n)$, and then calculates the value of $\mathbf{e}^T M(\varphi^n)\mathbf{a}$. This procedure, called Length, has a running time which, in general, is not bounded by a polynomial in $\log n$, $\log i$, w, and d. By making a few modifications, we improve Length's running time to achieve these bounds.

A problem arises immediately from multiplying $M(\varphi)$ by itself n times when we are only allowed polynomial time in $\log n$. The well-known *binary method of exponentiation* provides an efficient method for raising a matrix to the nth power. This method, — similar to the Russian peasant method of multiplication and presented by Knuth [Knu69, pg 399] — is reminiscent of the speedup Vitányi used for his D0L-membership algorithm [Vit72]. By repeatedly squaring $M(\varphi)$, we compute the matrices $M(\varphi)^{2^k}$ for $k \leq \log n$. Then computing $M(\varphi^n)$ involves only multiplying the appropriate matrices $M(\varphi)^{2^k}$ for values of k indicated by 1's in the binary expansion of n. Since there are at most $\lfloor \log_2 n \rfloor + 1$ of these matrices, we calculate $M(\varphi^n)$ with $O(\log n)$ matrix multiplications.

Our method so far has the flaw that the entries of the matrices potentially become too large, that is, not bounded by a polynomial in $\log n$, $\log i$, w, and d. We need this bound on matrix entries because the time it takes to perform arithmetic operations is polynomial in the number of bits of the operands.

Our first step towards bounding the entries of the matrices $M(\varphi^k)$ for $k \leq n$ entails bounding the entries of the columns corresponding to $\varphi^k(a)$. Consider the case where φ is *non-erasing*. After calculating any matrix $M(\varphi^k)$, we check if $\mathbf{e}^T M(\varphi^k)\mathbf{a} \geq i$. If so, we exit and report that $|\varphi^n(a)| \geq i$ because $|\varphi^k(a)| \leq |\varphi^n(a)|$ when $k \leq n$ and when φ is non-erasing.

However, if φ *is erasing*, then it is possible that $|\varphi^k(a)| \geq i$ and yet $|\varphi^n(a)| < i$, for some $k < n$. For example, consider the case where $\varphi(\mathtt{a}) = \mathtt{b}^{100}\mathtt{c}^{100}, \varphi(\mathtt{b}) = \mathtt{b}, \varphi(\mathtt{c}) = \epsilon$, and $i = 150$. Here, the calculation might stop prematurely since $|\varphi^1(\mathtt{a})| = 200$ but $|\varphi^n(\mathtt{a})| = 100$ for any $n > 1$. Consequently, reporting $|\varphi^n(a)| \geq i$ when $\mathbf{e}^T M(\varphi^k)\mathbf{a} \geq i$ may be incorrect.

We solve this by counting only the immortal letters in $\varphi^k(a)$. Let $\mathbf{v}$ be defined as the column vector

$$\mathbf{v} = (m_a)_{a\in\Sigma}, \text{ where } m_a = \begin{cases} 1, & \text{if } a \text{ is immortal;} \\ 0, & \text{otherwise.} \end{cases}$$

Rather than checking if $\mathbf{e}^T M(\varphi^k)\mathbf{a} \geq i$, we check if $\mathbf{v}^T M(\varphi^k)\mathbf{a} \geq i$. We will see in Lemma 4 that the number of mortal letters is not substantially greater than the number of immortal letters.

Up to now, we have focused on bounding the entries of the column corresponding to $\varphi^k(a)$ but we still have a problem if a is a slowly growing letter and other letters grow exponentially. For example, if we want to compute $\varphi^{100}(\mathtt{a})$ where $\varphi(\mathtt{a}) = \mathtt{a}, \varphi(\mathtt{b}) = \mathtt{bb}$, and $i = 20$, then

$$M(\varphi) = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix},$$

and

$$M(\varphi^{100}) = \begin{bmatrix} 1 & 0 \\ 0 & 2^{100} \end{bmatrix}.$$

In this simple case, we still want to compute that $|\varphi^{100}(\mathtt{a})| = 1$ without having to deal with $\varphi^{100}(\mathtt{b})$.

This problem of quickly growing letters is solved if we restrict Σ to the letters accessible from a. Modifying Σ this way has no effect on the results of Length, and as we will see in Lemma 5, it bounds the entries of $M(\varphi^k)$ in columns other than the one corresponding to $\varphi^k(a)$.

```
procedure POWER(φ, n, a, i)
// Returns M(φ^n) restricted to the letters accessible from a, making sure that the sum of
// the immortal letters of M(φ^n)a is less than i.

1    Y ← the identity matrix;
2    Z ← M(φ) restricted to the letters accessible from a;
3    v ← column vector indicating the immortal letters of φ;

     // calculate Y to be Z^n
4    while (n > 0) do
5        if (n ≡ 1 mod 2) then
6            Y ← YZ;
7            if (v^T Y a ≥ i) then return NIL;
8        n ← ⌊n/2⌋;
9        if (n > 0) then
10           Z ← Z^2;
11           if (v^T Z a ≥ i) then return NIL;

12   return Y;
```

Figure 3.1: The procedure POWER

3.1.1 Pseudocode for POWER and LENGTH

We are now ready to formally define LENGTH. Since the binary method of matrix exponentiation will be used for other calculations, we write the method as the subroutine POWER shown in Figure 3.1.

Provided POWER does not return NIL, POWER sets Y equal to Z^n with $O(\log n)$ matrix multiplications. If $\mathbf{v}^T Y\mathbf{a} \geq i$ at line 7, then

$$i \leq \mathbf{v}^T Y\mathbf{a} \leq \mathbf{v}^T M(\varphi^n)\mathbf{a} \leq \mathbf{e}^T M(\varphi^n)\mathbf{a} = |\varphi^n(a)|,$$

and therefore returning NIL is correct. Similarly, returning NIL at line 11 when $\mathbf{v}^T Z\mathbf{a} \geq i$ is also correct.

The procedure LENGTH (see Figure 3.2) calls POWER to construct $M(\varphi^n)$ restricted to

```
procedure LENGTH(φ, n, a, i)
// returns |φ^n(a)|, if |φ^n(a)| < i; −1, otherwise.

    e ← column vector of all 1's;
    a ← Parikh vector of a;

    // calculate M(φ^n) restricted to the letters accessible from a
    Z ← POWER(φ, n, a, i);

    // return the result
    if ((Z = NIL) or (e^T Z a ≥ i)) then return −1;
    return e^T Z a;
```

Figure 3.2: The procedure LENGTH

the letters accessible from a. If the result is not NIL, and $\mathbf{e}^T M(\varphi^n)\mathbf{a} < i$, then LENGTH returns $\mathbf{e}^T M(\varphi^n)\mathbf{a} = |\varphi^n(a)|$. We assume $n \geq 0$, and $i \geq 1$.

3.1.2 Analysis of POWER and LENGTH

Before discussing the running time of LENGTH, we prove a theorem about the size of the numbers that POWER deals with. Theorem 6 below shows that by performing the checks on lines 7 and 11 of POWER, the number of bits needed to store any matrix entry during the execution of POWER is bounded by a polynomial in $\log n$, $\log i$, w, and d. We start with two technical lemmas.

In the following lemma, let Im be the function that takes a word and deletes the mortal letters from it.

Lemma 4 *If φ, a, i, w, and d are defined as before, and $|\mathrm{Im}(\varphi^k(a))| \leq i$ for some integer $k > 0$, then $|\varphi^k(a)| \leq w^d i$.*

Proof. If $k \leq d$, then $|\varphi^k(a)| \leq w^k \leq w^d$ and so the statement is true for any i. Hence we assume $k > d$.

Applying φ^d to a mortal letter produces the empty string. Then applying φ^d to $\varphi^{k-d}(a)$

is equivalent to applying φ^d to the immortal letters of $\varphi^{k-d}(a)$. More precisely,

$$\varphi^k(a) = \varphi^d(\mathrm{Im}(\varphi^{k-d}(a)).$$

On the other hand, applying φ^d to an immortal letter produces a word of length at most w^d. Thus,

$$|\varphi^k(a)| \leq w^d |\mathrm{Im}(\varphi^{k-d}(a))|.$$

Notice $|\mathrm{Im}(\varphi^{k-d}(a))| \leq |\mathrm{Im}(\varphi^k(a))| \leq i$. Therefore,

$$|\varphi^k(a)| \leq w^d i.$$

■

Lemma 5 *If φ, a, i, w, and d are defined as before, and b is accessible from a, then $|\varphi^k(b)| \leq w^d |\varphi^k(a)|$.*

Proof. Since b is a letter accessible from a, then for some $\ell \leq d$, $\varphi^\ell(a)$ contains a b. Hence $\varphi^k(b)$ is a subword of $\varphi^k(\varphi^\ell(a))$ and therefore,

$$|\varphi^k(b)| \leq |\varphi^\ell(\varphi^k(a))|.$$

Applying φ^ℓ to any word increases the length by at most a factor of w^ℓ. Hence,

$$|\varphi^k(b)| \leq w^\ell |\varphi^k(a)|.$$

And since $\ell \leq d$,

$$|\varphi^k(b)| \leq w^d |\varphi^k(a)|.$$

■

We now prove our bounds on the entries of incidence matrices calculated by POWER.

Theorem 6 *If $M(\varphi^\ell)$ is a matrix calculated during the execution of* POWER, *then each entry of $M(\varphi^\ell)$ is less than $dw^{4d}i^2$.*

Proof. If $\ell = 1$ then each entry of $M(\varphi^\ell)$ is bounded by w.

Otherwise $M(\varphi^\ell)$ is calculated by multiplying $M(\varphi^j)$ and $M(\varphi^k)$ where $\mathbf{v}^T M(\varphi^j)\mathbf{a} \leq i$ and $\mathbf{v}^T M(\varphi^k)\mathbf{a} \leq i$. By Lemma 4, $\mathbf{e}^T M(\varphi^k)\mathbf{a} \leq w^d i$, and then by Lemma 5, the sum of any column of $M(\varphi^k)$ is less than $w^{2d}i$. By a similar argument the sum of any column of $M(\varphi^j)$ is less than $w^{2d}i$ and hence the sum of any row in $M(\varphi^j)$ is less than $dw^{2d}i$. Therefore, each entry in $M(\varphi^\ell)$ is bounded by $w^{2d}i \cdot dw^{2d}i = dw^{4d}i^2$. ∎

Finally, we discuss the running time of LENGTH.

Claim 7 *The procedures* POWER *and* LENGTH *run in $O((\log n)d^3(\log d + d\log w + \log i)^2)$ bit operations.*

Proof. The calculation of $\mathbf{e}^T Z\mathbf{a}$ in LENGTH takes less time than the matrix multiplications in POWER so we restrict our attention to the running time of POWER.

By Theorem 6, no entry of Y and Z in POWER is greater than $dw^{4d}i^2$. Hence, we need to do $O(\log n)$ multiplications of $d \times d$ matrices with entries of size $\leq O(\log d + d\log w + \log i)$ bits. The total time taken by the matrix multiplications is $O((\log n)d^3(\log d + d\log w + \log i)^2)$.

The checks at lines 7 and 11 of POWER which are done after each matrix multiplication involves at most d^2 multiplications and additions, which requires only $O(d^2(\log d + d\log w + \log i)^2)$ bit operations for each check.

Hence, POWER runs in $O((\log n)d^3(\log d + d\log w + \log i))$ bit operations. ∎

3.2 GenericLength

It will be useful to find the length of words of the form

$$\varphi^{tq+r}(x)\varphi^{(t+1)q+r}(x)\cdots\varphi^{(s-1)q+r}(x),$$

for non-negative integers q, r, s and t, where $q > r$ and for some word x. We calculate the lengths of these words using a function more generic than Length called GenericLength.

Formally, GenericLength solves the following problem:

Problem 8 *Given a homomorphism $\varphi : \Sigma^* \to \Sigma^*$ of width w and depth d, the Parikh vector $\mathbf{x}$ of a word $x \in \Sigma^*$, non-negative integers q, r, s, t, and i such that $r < q \le d$, then return*

$$\begin{cases} \sum_{t \le j < s} |\varphi^{jq+r}(x)|, \text{ if } \sum_{t \le j < s} |\varphi^{jq+r}(x)| < i; \\ -1, \text{ otherwise.} \end{cases}$$

in time polynomial in $\log n$, $\log i$, w, and d, where $n = sq + r$.

Although GenericLength calculates the same result that Length does when $s-1 = t$, and $|x| = 1$, Length is still useful because it is straightforward and quicker for such cases.

3.2.1 SuperPower

An instrumental idea used in GenericLength is the ability to calculate matrices of the form $\mathbf{C}_n = \mathbf{I} + \mathbf{A} + \cdots + \mathbf{A}^{n-1}$ efficiently. The following lemma suggests a method for calculating $\mathbf{C}_n$.

Lemma 9 *If $\mathbf{A}$ is a square matrix, and n is an integer ≥ 1, let $\mathbf{B} = \begin{bmatrix} \mathbf{A} & \mathbf{O} \\ \mathbf{I} & \mathbf{I} \end{bmatrix}$, where $\mathbf{O}$ and $\mathbf{I}$ are the zero and identity matrix of the same size as $\mathbf{A}$. Then $\mathbf{B}^n = \begin{bmatrix} \mathbf{A}^n & \mathbf{O} \\ \mathbf{C}_n & \mathbf{I} \end{bmatrix}$, where $\mathbf{C}_n = \mathbf{I} + \mathbf{A} + \cdots + \mathbf{A}^{n-1}$.*

Proof. By induction on n.

The statement is true for the base case of $n = 1$ by definition of $\mathbf{B}$.

Assume the statement holds for $n - 1$. Hence,

$$\mathbf{B}^{n-1}\mathbf{B} = \begin{bmatrix} \mathbf{A}^{n-1} & \mathbf{O} \\ \mathbf{I} + \mathbf{A} + \cdots + \mathbf{A}^{n-2} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{A} & \mathbf{O} \\ \mathbf{I} & \mathbf{I} \end{bmatrix}.$$

Therefore,

$$\mathbf{B}^{n} = \begin{bmatrix} \mathbf{A}^{n} & \mathbf{O} \\ \mathbf{I} + \mathbf{A} + \cdots + \mathbf{A}^{n-1} & \mathbf{I} \end{bmatrix} = \begin{bmatrix} \mathbf{A}^{n} & \mathbf{O} \\ \mathbf{C}_n & \mathbf{I} \end{bmatrix}.$$

■

Using the binary method of exponentiation, and Lemma 9, we construct a subroutine to compute $\mathbf{C}_{n+1}$ called SuperPower shown in Figure 3.3. Similar to Power, SuperPower performs a check after each new matrix is calculated. As we will see in Theorem 10 below, these checks ensure that the entries do not exceed a polynomial in $\log n$, $\log i$, w, and d.

3.2.2 The Running Time of SuperPower

Before discussing the running time of SuperPower, we prove a theorem bounding the entries of any matrix calculated by SuperPower.

Theorem 10 *If $\varphi : \Sigma^* \to \Sigma^*$ is a homomorphism of depth d and width w, and $\mathbf{A} = M(\varphi^k)$ restricted to the letters accessible from a non-empty word x for some k, and $\mathbf{B}^\ell$ is a matrix calculated by* SuperPower$(\mathbf{A}, n, \mathbf{x}, i)$*, then each entry of $\mathbf{B}^\ell$ is less than $2dw^{8dk}i^2$.*

Proof. The matrix $\mathbf{B} = \left[\begin{smallmatrix} \mathbf{A} & \mathbf{O} \\ \mathbf{I} & \mathbf{I} \end{smallmatrix}\right]$, is the same as the incidence matrix for the homomorphism $\varphi' : \Sigma'^* \to \Sigma'^*$, where $\Sigma' = \{a_1, a_2, \cdots, a_d, a'_1, a'_2, \cdots, a'_d\}$, and φ' is defined by: $\varphi'(a_j) = \varphi^k(a_j)$, and $\varphi'(a'_j) = a_j a'_j$ for all $j \le d$.

SuperPower computes $M(\varphi'^n)$ similar to the way Power computes $M(\varphi^n)$ but with a few differences. Namely, there is no regard for which letters are immortal or for which letters are accessible from x. We address these issues first.

procedure SUPERPOWER($\mathbf{A}, n, \mathbf{x}, i$)
// returns $\mathbf{C}_{n+1} = \mathbf{I} + \mathbf{A} + \cdots + \mathbf{A}^n$, making sure that the sum of the entries of $\mathbf{C}_{n+1}\mathbf{x}$
// is less than i

$\mathbf{B} \leftarrow \left[\begin{smallmatrix} \mathbf{A} & \mathbf{O} \\ \mathbf{I} & \mathbf{I} \end{smallmatrix}\right]$;
$\mathbf{D} \leftarrow$ the identity matrix of the same order as $\mathbf{B}$;
$\mathbf{x}' \leftarrow \left[\begin{smallmatrix} \mathbf{x} \\ \mathbf{0} \end{smallmatrix}\right]$;
$\mathbf{e}' \leftarrow$ column vector of 1's of the same order as $\mathbf{x}'$;

// calculate $\mathbf{D}$ to be $\mathbf{B}^n$
while ($n > 0$) do
    if ($n \equiv 1 \bmod 2$) then
        $\mathbf{D} \leftarrow \mathbf{D}\mathbf{B}$;
        if ($\mathbf{e}'^T\mathbf{D}\mathbf{x}' \geq i$) then return NIL;
    $n \leftarrow \lfloor n/2 \rfloor$;
    if ($n > 0$) then
        $\mathbf{B} \leftarrow \mathbf{B}^2$;
        if ($\mathbf{e}'^T\mathbf{B}\mathbf{x}' \geq i$) then return NIL;

//$\mathbf{C}_n$ is in the lower left quadrant of $\mathbf{D}$ and
//$\mathbf{A}^n$ is in the upper left quadrant of $\mathbf{D}$
return ($\mathbf{C}_n + \mathbf{A}^n$);

Figure 3.3: The procedure SUPERPOWER

Since $\mathbf{e}'^T\mathbf{B}^\ell\mathbf{x}' = \mathbf{e}^T\mathbf{C}_{\ell+1}\mathbf{x}$, then $\mathbf{e}'^T\mathbf{B}^\ell\mathbf{x}$ is non-decreasing with respect to ℓ. Therefore, SUPERPOWER correctly returns NIL when $\mathbf{e}'^T\mathbf{B}^\ell\mathbf{x}' \geq i$ without any problems caused by the deletion of mortal letters.

By our assumption, the set of letters $\{a_1, a_2, \cdots, a_d\}$ is accessible from x and we see that $\{a'_1, a'_2, \cdots, a'_d\}$ is not. However, as Lemma 9 shows, the entries in $\mathbf{B}^\ell$ corresponding to these inaccessible letters are either 0 or 1 and therefore in no danger of growing too large.

Now that the above concerns are handled, an argument analogous to Theorem 6 shows that each entry of $\mathbf{B}^\ell$ is less than $d'w'^{4d'}i^2$, where w' and d' are the width and depth of φ' respectively. Since $w' \leq w^k$ and $d' = 2d$, then each entry of $\mathbf{B}^\ell$ is less than $(2d)(w^k)^{4(2d)}i^2 = 2dw^{8dk}i^2$. ∎

Claim 11 SUPERPOWER$(M(\varphi^k), n, \mathbf{x}, i)$ *runs in* $O((\log n)d^3(\log d + dk \log w + \log i)^2)$ *bit operations.*

We follow the proof of Claim 7 closely.

Proof. By Theorem 10, the entries of any matrix are smaller than $2dw^{8dk}i^2$. Hence, we need to do $O(\log n)$ multiplications of $2d \times 2d$ matrices with entries of size $\leq O(\log d + dk \log w + \log i)$ bits. The total time taken by the matrix multiplications is $O((\log n)d^3(\log d + dk \log w + \log i)^2)$.

The checks at lines 8 and 12 of SUPERPOWER which are done after each matrix multiplication involve at most $O(d^2)$ multiplications and additions, which requires only $O(d^2(\log d + dk \log w + \log i))$ bit operations for each check.

Hence, SUPERPOWER runs in $O((\log n)d^3(\log d + dk \log w + \log i)^2)$ bit operations. ∎

3.2.3 Analysis of GENERICLENGTH

GENERICLENGTH acts differently depending upon the values of s and t.

procedure GenericLength($\varphi, \mathbf{x}, q, r, s, t, i$)
// returns $\sum_{t \le j < s} |\varphi^{jq+r}(x)|$, if $\sum_{t \le j < s} |\varphi^{jq+r}(x)| < i$.
// returns -1, otherwise.
$\mathbf{e} \leftarrow$ column vector of all 1's;

if $(s - 1 < t)$ then
  return 0;

else if $(s - 1 = t = 0)$ then
  $Z \leftarrow$ Power($\varphi, r, \mathbf{x}, i$); $//M(\varphi^r)$
  if ($Z =$ nil) then return -1;

else
  $Y \leftarrow$ Power($\varphi, tq + r, \mathbf{x}, i$); $//M(\varphi^{tq+r})$
  if ($Y =$ nil) then return -1;
  $Z \leftarrow$ Power($\varphi, q, \mathbf{x}, i$); $//M(\varphi^q)$
  if ($Z =$ nil) then return -1;
  $Z \leftarrow$ SuperPower($Z, s - t - 1, \mathbf{x}, i$); $//M(\varphi^{(0)q}) + \cdots + M(\varphi^{(s-t-1)q})$
  if ($Z =$ nil) then return -1;
  $Z \leftarrow YZ$; $//M(\varphi^{tq+r}) + \cdots + M(\varphi^{(s-1)q+r})$

if $(\mathbf{e}^T Z\mathbf{x} \ge i)$ then return -1;
return $\mathbf{e}^T Z\mathbf{x}$;

Figure 3.4: The procedure GenericLength

If $s \le t$ then GenericLength returns 0, signifying that no numbers are less than s and greater than or equal to t.

If $s - 1 = t$, then we simply calculate the matrix $Z = M(\varphi^{qt+r})$ using Power. This saves GenericLength from calculating $M(\varphi^q)$ when calculating only $M(\varphi^r)$ may be necessary, that is, when $t = 0$.

Otherwise GenericLength performs a series of matrix operations to compute the matrix $Z = M(\varphi^{tq+r}) + M(\varphi^{(t+1)q+r}) + \cdots + M(\varphi^{(s-1)q+r})$.

In the latter two cases, GenericLength uses Z and $\mathbf{x}$ to obtain the final result. (see Figure 3.4)

Claim 12 GenericLength *returns* $\sum_{t \le j < s} |\varphi^{jq+r}(x)|$, *if* $\sum_{t \le j < s} |\varphi^{jq+r}(x)| < i$; -1, *oth-*

erwise.

If $s - 1 = t = 0$, then line 5 simply calculates $Z = M(\varphi^r)$. Hence line 16 returns $\mathbf{e}^T M(\varphi^r)\mathbf{x} = |\varphi^r(w)| = \sum_{j=0} |\varphi^{jq+r}(x)|$, which is correct.

Otherwise, GENERICLENGTH calls POWER to calculate the matrices $Y = M(\varphi^{tq+r})$, and $Z = M(\varphi^q)$, and then it calls SUPERPOWER to calculate

$$\begin{aligned} Z &= Z^0 + Z^1 + \cdots + Z^{s-t-1} \\ &= M(\varphi^q)^0 + M(\varphi^q)^1 + \cdots + M(\varphi^q)^{s-t-1} \\ &= M(\varphi^0) + M(\varphi^q) + \cdots + M(\varphi^{(s-t-1)q}). \end{aligned}$$

The final value for Z is calculated as:

$$\begin{aligned} Z &= YZ \\ &= \left(M(\varphi^{tq+r})\right) \left(M(\varphi^0) + M(\varphi^1) + \cdots + M(\varphi^{(s-t-1)q})\right) \\ &= M(\varphi^{tq+r}) + M(\varphi^{(t+1)q+r}) + \cdots + M(\varphi^{(s-1)q+r}). \end{aligned}$$

Therefore, Line 16 returns

$$\begin{aligned} \mathbf{e}^T Z\mathbf{x} &= \mathbf{e}^T \left(M(\varphi^{tq+r}) + \cdots + M(\varphi^{(s-1)q+r})\right) \mathbf{x} \\ &= \mathbf{e}^T M(\varphi^{tq+r})\mathbf{x} + \cdots + \mathbf{e}^T M(\varphi^{(s-1)q+r})\mathbf{x} \\ &= |\varphi^{tq+r}(x)| + \cdots + |\varphi^{(s-1)q+r}(x)| \\ &= \textstyle\sum_{t \le j < s} |\varphi^{jq+r}(x)|. \end{aligned}$$

We finish the proof by showing that GENERICLENGTH is correct for the cases where it exits and returns -1 before line 16. We concern ourselves with any intermediate calculations of Z. We need to ensure that, if NIL is returned by POWER or SUPERPOWER, then GENERICLENGTH does not return -1 when $\sum_{t \le j < s} |\varphi^j(x)| < i$.

If POWER returns NIL for the case where $s - 1 = 0$ (line 5), then we know $\mathbf{e}^T M(\varphi^r)\mathbf{x} \ge i$. Thus, returning -1 at line 6 is correct.

Now consider the case handled by lines 7 to 14 where $s-1>0$. Thus, $(s-1)q+r>q$ when POWER$(\varphi, q, \mathbf{x}, i)$ is called at line 10. Consequently,the number of immortal letters in $\varphi^q(x)$ is less than or equal to the number of letters in $\varphi^{(s-1)q+r}$. Hence, if POWER returns NIL at line 10, then GENERICLENGTH is correct in returning -1.

If POWER$(\varphi, tq+r, \mathbf{x}, i)$ returns NIL at line 8, then it is safe to return -1 since the matrix $M(\varphi^{tq+r})$ is a factor in the final calculation of Z. Similarly, the matrix returned by SUPERPOWER on line 12, is also a factor in the final calculation of Z, Hence, returning -1 if SUPERPOWER returns NIL is also correct. ■

We end this section by proving a theorem about the running time of GENERICLENGTH.

Claim 13 GENERICLENGTH *runs in* $O((\log n)d^3(\log d + d^2 \log w + \log i)^2)$ *time.*

Proof. The longest possible execution of GENERICLENGTH calculates the matrix Z by stepping through lines 9 to 14. Recall that $n = sq+r$. At line 9, $t<s$, and hence $qt+r<n$. Therefore, by Claim 7, the running time of lines 8 and 10 are $O((\log n)d^3(\log d + d \log w + \log i)^2)$.

By Claim 11, the running time of SUPERPOWER is $O((\log n)d^3(\log d + dk \log w + \log i)^2)$ where $k=q$. By the statement of the problem, $q \leq d$.

Hence the running time of SUPERPOWER is $O((\log n)d^3(\log d + d^2 \log w + \log i)^2)$. ■

3.3 BASICTREEDESCENT

The final subroutine we discuss is a simple method to calculate the ith letter of $\varphi^n(a)$, called BASICTREEDESCENT. This algorithm is equivalent to one found by Jones and Skyum [JS81]. While its running time is *not* necessarily polynomial in $\log n$, $\log i$, w, and d, we use BASICTREEDESCENT to calculate the ith letter of $\varphi^n(a)$ for the cases in which n *is* polynomial in $\log i$, w, and d.

Instead of computing $\varphi^n(a)$ by repeatedly applying φ to each successive word, we calculate the ith letter of $\varphi^n(a)$ by descending n levels down the derivation tree of $\varphi^n(a)$ to the appropriate letter. The only difficulty lies in determining which subtree to descend.

Let $\varphi(a) = z = z_1 z_2 \cdots z_{|\varphi(a)|}$. BasicTreeDescent uses Length to compute

$$|\varphi^{n-1}(z_1)|, |\varphi^{n-1}(z_2)|, \cdots$$

and by adding, computes

$$|\varphi^{n-1}(z_1)|, |\varphi^{n-1}(z_1 z_2)|, \cdots$$

until a value t is determined such that

$$|\varphi^{n-1}(z_1 z_2 \cdots z_{t-1})| < i \leq |\varphi^{n-1}(z_1 z_2 \cdots z_t)|.$$

Once t has been calculated, the algorithm adjusts the values of n, a, and i to $n-1$, z_t, and $i - |\varphi^{n-1}(z_1 z_2 \cdots z_{t-1})|$ respectively and thus descends one level in the derivation tree. Of course, if $n = 0$, the algorithm simply returns a.

Figure 3.5 gives the algorithm for BasicTreeDescent.

As an example, consider the case where BasicTreeDescent$(\varphi, n, \mathtt{a}, i)$ is called such that $\varphi(\mathtt{a}) = \mathtt{bcd}$ and $|\varphi^{n-1}(\mathtt{b})| < i \leq |\varphi^{n-1}(\mathtt{bc})|$, then the first step of the subroutine descends down the branch with `c` as its root. See Figure 3.6 for an illustration.

During the execution of the Basic Tree Descent algorithm, we descend exactly n levels. Each time we do so, we calculate the length of at most w words. Since the running time is at least linear in n, BasicTreeDescent is only useful to us when n itself is bounded by a polynomial in $\log i, w$, and d. Theorem 14 gives the running time of BasicTreeDescent

Theorem 14 *The procedure* BasicTreeDescent *uses* $O(nw(\log n)d^3(d \log w + \log i)^2)$ *bit operations.*

Proof. We obtain the result by multiplying the running time of Length by nw. ∎

```
procedure BasicTreeDescent(φ, n, a, i)
// returns the ith letter of φ^n(a) using n steps.
    while (n > 0) do
        z ← φ(a);
        total ← 0;

        for (t ← 1 to z) do
            len ← Length(n − 1, z_t, i);
            total ← total + len;
            if (total ≥ i) or (len = −1); then
                total ← total − len;
                break from for-loop;

        n ← n − 1;
        a ← z_t;
        i ← i − total;

    return a;
```

Figure 3.5: The procedure BasicTreeDescent

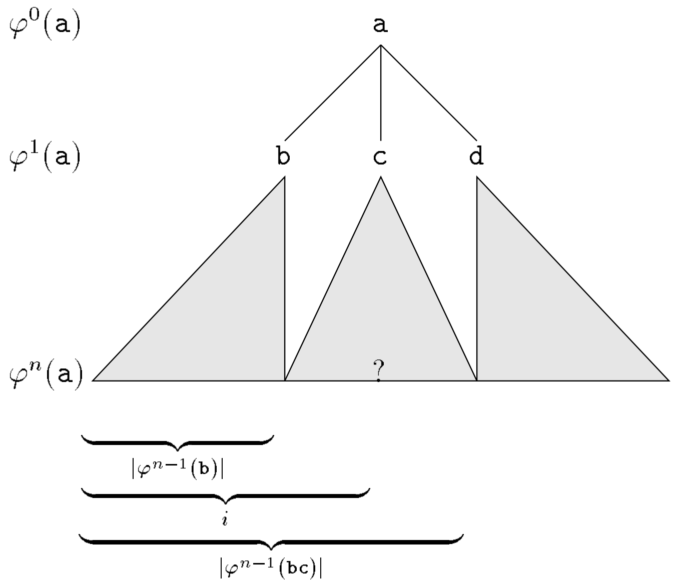

Figure 3.6: $|\varphi^{n-1}(\mathtt{b})| < i \leq |\varphi^{n-1}(\mathtt{bc})|$

Chapter 4

The FIND Algorithm

4.1 The Algorithm

The idea behind our new algorithm for computing the ith letter of $\varphi^n(a)$ is to avoid descending n levels in the derivation tree for the cases where n is not bounded by a polynomial in $\log i$, w, and d, by exploiting the recursive nature of the homomorphism. Our approach is reminiscent of the pumping lemma for CFL's [HU69, pp. 125–130]. Basically, we descend the derivation tree until we encounter a letter for the second time and then shortcut to the last occurrence of this repeated letter in our descent to the ith letter. We continue descending, taking shortcuts whenever possible until we reach the ith letter of the bottom level.

4.1.1 Finding a Repeated Letter

We discuss some notation concerning the derivation path taken. Let a_0 be the root of the derivation tree. Let a_j be the letter encountered after descending j levels and set x_j, y_j such that $\varphi(a_{j-1}) = x_j a_j y_j$. Therefore, after descending ℓ levels, the sequence of letters encountered during the tree descent is $a_0, a_1, \ldots, a_\ell$ and the sequences of words $x_1, x_2, \ldots, x_\ell$ and $y_1, y_2, \ldots, y_\ell$ describe the branches of the tree which were not followed.

We begin our algorithm by descending along the derivation tree until we encounter some letter a_ℓ for the second time, where ℓ is the number of levels we have descended thus far. Let $\ell - q$ be the level containing the previous occurrence of a_ℓ. We need only descend d levels before encountering such a letter a_ℓ since $|\Sigma| = d$. More precisely, $a_\ell = a_{\ell-q}$ for some $\ell \le d$; see Figure 4.1.

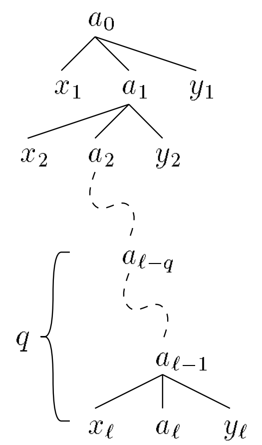

Figure 4.1: Descending to a repeated letter a_ℓ.

By descending ℓ levels, we have calculated new values for n, a, and i, which yield the same results, namely $n - \ell$, a_ℓ, and $i - |\varphi^{n-1}(x_1)\varphi^{n-2}(x_2)\cdots\varphi^{n-\ell}(x_\ell)|$. We have also discovered words x and y and an integer q such that $\varphi^q(a) = xay$. Specifically, x and y are $\varphi^{q-1}(x_{\ell-q+1})\cdots\varphi(x_{\ell-1})\, x_\ell$, and $y_\ell\, \varphi(y_{\ell-1})\cdots\varphi^{q-1}(y_{\ell-q+1})$ respectively.

4.1.2 Jumping Ahead

Once a repeated letter a is found, the remainder of the algorithm calculates the last occurrence of a in our descent to the ith letter and then either invokes the BasicTreeDescent algorithm or restarts our algorithm with new values of n and i.

Setting s and r such that $n = sq+r$ and $r < q$, we use the words x, y, and the integers q, r, and s to describe $\varphi^n(a)$. We know $\varphi^q(a) = x\,a\,y$ and $\varphi^{2q}(a) = \varphi^q(x)\,x\,a\,y\,\varphi^q(y)$. Continuing this way, we see that

$$\varphi^{sq}(a) = \varphi^{(s-1)q}(x) \cdots \varphi^{q}(x)\, x\, a\, y\, \varphi^{q}(y) \cdots \varphi^{(s-1)q}(y).$$

Finally, applying φ^r to both sides yields

$$\varphi^{n}(a) = \underbrace{\varphi^{(s-1)q+r}(x) \cdots \varphi^{q+r}(x)\, \varphi^{r}(x)}_{X_1}\ \underbrace{\varphi^{r}(a)}_{X_2}\ \underbrace{\varphi^{r}(y)\, \varphi^{q+r}(y) \cdots \varphi^{(s-1)q+r}(y)}_{X_3}.$$

Using GenericLength and Length, we determine which of X_1, X_2, or X_3 the ith letter falls in and we handle these three cases separately.

Case 1: X_1 contains the ith letter. If x contains an exponentially growing letter, then Lemma 16 below states that $n = O(d \log i)$ and therefore, we safely call BasicTreeDescent.

Otherwise, x grows polynomially. Using GenericLength, we do a binary search for a value of t such that the ith letter lies in the subword $\varphi^{tq+r}(x)$, more precisely,

$$|\varphi^{(s-1)q+r}(x) \cdots \varphi^{(t+1)q+r}(x)| < i \leq |\varphi^{(s-1)q+r}(x) \cdots \varphi^{tq+r}(x)|.$$

Once t is found, we set $n \leftarrow (t+1)q + r$ and $i \leftarrow i - |\varphi^{(s-1)q+r}(x) \cdots \varphi^{(t+1)q+r}(x)|$. As we prove in Lemma 17 below, adjusting the values of i and n this way lets us descend to the last occurrence of a in the derivation path to the ith letter.

Case 2: X_2 contains the ith letter. Since we have already calculated the length of X_1, we need only set $i \leftarrow i - |X_1|$, set $n \leftarrow r$, and then call BasicTreeDescent. We call BasicTreeDescent, since now $n = r < q \leq d$.

Case 3: X_3 contains the ith letter. Similarly to Case 1, we use GenericLength to do a binary search for a value t, such that the ith letter lies in the the subword $\varphi^{tq+r}(y)$, that is,

$$|\varphi^r(y)\varphi^{q+r}(y)\cdots\varphi^{(t-1)q+r}(y)| < i - |X_1X_2| \leq |\varphi^r(y)\varphi^{q+r}(y)\cdots\varphi^{tq+r}(y)|.$$

Once the value of t is found, we rewrite $\varphi^n(a)$ as

$$\varphi^{(s-1)q+r}(x)\cdots\varphi^{(t+1)q+r}(x)\varphi^{(t+1)q+r}(a)\varphi^{(t+1)q+r}(y)\cdots\varphi^{(s-1)q+r}(y).$$

Since we wish to reduce i by the number of letters to the *left* of the subword $\varphi^{(t+1)q+r}(a)$, we set $i \leftarrow i - |\varphi^{(s-1)q+r}(x)\cdots\varphi^{(t+1)q+r}(x)|$ and $n \leftarrow (t+1)q+r$.

If y is polynomially growing, then Lemma 17 below shows that the letter a is not encountered again. Hence, we continue descending along the tree until the next shortcut is taken or until the bottom is reached.

Otherwise, y grows exponentially. By Lemma 16 below, $n = O(d \log i)$ and therefore we call BasicTreeDescent.

Figure 4.2 gives the algorithm Find which finds the ith letter of $\varphi^n(a)$. We assume that $1 \leq i \leq |\varphi^n(a)|$.

procedure $\textsc{Find}(\varphi, n, a, i)$;
$a_0 \leftarrow a$;

for $(\ell \leftarrow 1$ to $\infty)$ //executes for $\leq d$ iterations
  $z \leftarrow \varphi(a) = z_1 z_2 \cdots z_{|\varphi(a)|}$;
  find smallest index t such that $i > |\varphi^{n-1}(z_1 z_2 \cdots z_{t-1})|$;
  $i \leftarrow i - |\varphi^{n-1}(z_1 z_2 \cdots z_{t-1})|$; $n \leftarrow n - 1$; $a \leftarrow z_t$;
  $x_\ell \leftarrow z_1 \cdots z_{t-1}$; $a_\ell \leftarrow z_t$; $y_\ell \leftarrow z_{t+1} \cdots z_{|\varphi(a)|}$;

  if $(n \leq d)$ then return $\textsc{BasicTreeDescent}(\varphi, n, a, i)$;
  else if $(a_j = a_\ell$ for some $j < \ell)$ then
    $q \leftarrow \ell - j$; $s \leftarrow \lfloor n/q \rfloor$; $r \leftarrow n - sq$;
    $\mathbf{x} \leftarrow$ Parikh vector of $\varphi^{q-1}(x_{\ell-q+1}) \cdots \varphi(x_{\ell-1})\, x_\ell$;
    $\mathbf{y} \leftarrow$ Parikh vector of $y_\ell\, \varphi(y_{\ell-1}) \cdots \varphi^{q-1}(y_{\ell-q+1})$;
    $S_1 \leftarrow \textsc{GenericLength}(\varphi, \mathbf{x}, q, r, s, 0, i)$; //Length of X_1.
    $S_2 \leftarrow \textsc{Length}(\varphi, r, a, i - S_1)$; //Length of X_2.

    if $(i \leq S_1)$ then //Case 1
      if (x grows exponentially) then return $\textsc{BasicTreeDescent}(\varphi, n, a, i)$;
      use $\textsc{GenericLength}$ to do a binary search for t such that
        $|\varphi^{(s-1)q+r}(x) \cdots \varphi^{(t+1)q+r}(x)| < i \leq |\varphi^{(s-1)q+r}(x) \cdots \varphi^{tq+r}(x)|$
      $i \leftarrow i -$ $\textsc{GenericLength}(\varphi, \mathbf{x}, q, r, s, t+1, i)$;
      $n \leftarrow (t+1)q + r$;
      return $\textsc{Find}(\varphi, n, a, i)$;
    else if $(i \leq S_1 + S_2)$ then //Case 2
      return $\textsc{BasicTreeDescent}(\varphi, r, a, i - S_1)$;
    else // Case 3
      use $\textsc{GenericLength}$ to do a binary search for t such that
        $|\varphi^r(y)\varphi^{q+r}(y) \cdots \varphi^{(t-1)q+r}(y)| < i - S_1 - S_2 \leq |\varphi^r(y)\varphi^{q+r}(y) \cdots \varphi^{tq+r}(y)|$
      $i \leftarrow i -$ $\textsc{GenericLength}(\varphi, \mathbf{x}, q, r, s, t+1, i)$;
      $n \leftarrow (t+1)q + r$;
      if (y grows exponentially) then return $\textsc{BasicTreeDescent}(\varphi, n, a, i)$;
      else return $\textsc{Find}(\varphi, n, a, i)$;

Figure 4.2: The procedure FIND

4.2 Some Examples

We step through some examples in order to demonstrate various aspects of our algorithm. Consider the homomorphism φ defined as $\mathtt{a} \to \mathtt{bec}$, $\mathtt{b} \to \mathtt{bd}$, $\mathtt{c} \to \mathtt{cc}$, $\mathtt{d} \to \mathtt{d}$, and $\mathtt{e} \to \mathtt{bac}$.

Note that $\mathtt{a}$, $\mathtt{c}$, and $\mathtt{e}$ grow exponentially and $\mathtt{b}$ and $\mathtt{d}$ grow polynomially. We consider the calculations of the 495th, 497th, and the 499th letters of the word $\varphi^{31}(\mathtt{a})$. Figure 4.3 concentrates on the parts of the derivation tree which are relevant to our examples.

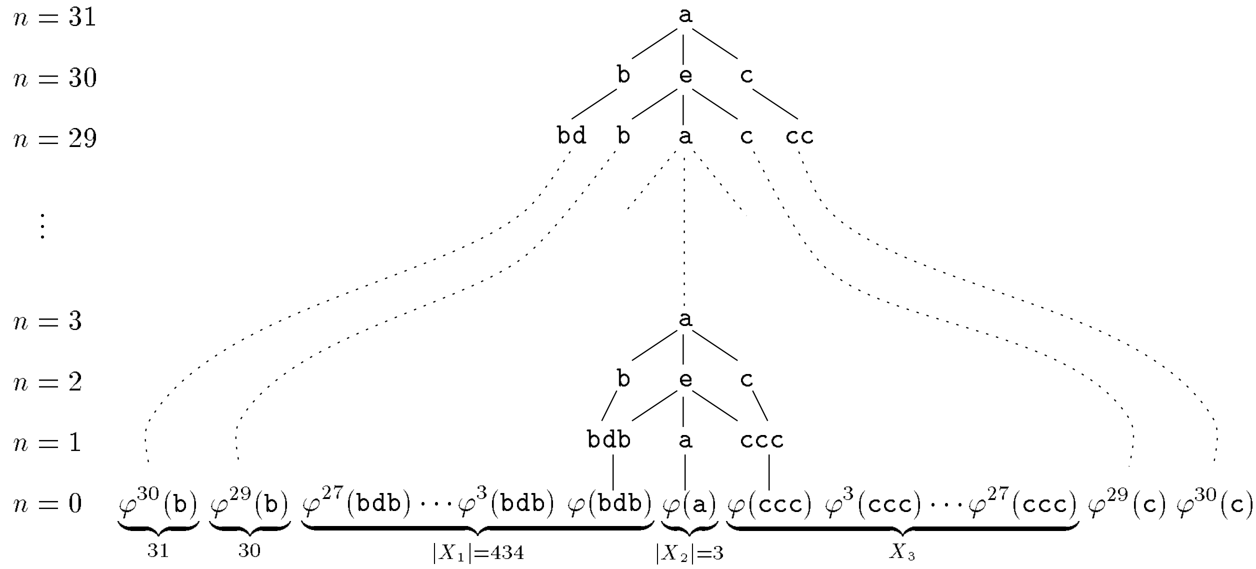

Figure 4.3: The derivation tree for $\varphi^{31}(\mathtt{a})$

Case a: $i = 495$. Find begins by calculating that $|\varphi^{30}(\mathtt{b})| < 495 \leq |\varphi^{30}(\mathtt{be})|$. Therefore, we descend one level by setting $n \leftarrow 30$, $i \leftarrow i - |\varphi^{30}(\mathtt{b})| = 464$, and $a \leftarrow \mathtt{e}$. We continue descending since we have not found a repeated letter. Since $|\varphi^{29}(\mathtt{b})| < 464 \leq |\varphi^{29}(\mathtt{ba})|$, we descend to the second level by setting $n \leftarrow 29$, $i \leftarrow i - |\varphi^{29}(\mathtt{b})| = 434$, and $a \leftarrow \mathtt{a}$.

Thus a repeated letter $\mathtt{a}$ is found with $q = 2$ representing the distance between both $\mathtt{a}$'s on the derivation path. We want s and r such that $n = qs + r$. We also want x and y such that $\varphi^2(\mathtt{a}) = x\mathtt{a}y$; Hence, $s \leftarrow 14$, $r \leftarrow 1$, $x \leftarrow \mathtt{bdb}$, and $y \leftarrow \mathtt{ccc}$.

We now write $\varphi^{29}(\mathtt{a})$ as $X_1X_2X_3$, where

$$\begin{aligned} X_1 &= \varphi^{(s-1)q+r}(x)\varphi^{(s-2)q+r}(x)\cdots\varphi^{r}(x) \\ &= \varphi^{27}(\mathtt{bdb})\varphi^{25}(\mathtt{bdb})\cdots\varphi(\mathtt{bdb}), \\ X_2 &= \varphi^{r}(a) \\ &= \mathtt{bec}, \\ X_3 &= \varphi^{r}(y)\varphi^{q+r}(y)\cdots\varphi^{(s-1)q+r}(y) \\ &= \varphi(\mathtt{ccc})\varphi^{3}(\mathtt{ccc})\cdots\varphi^{27}(\mathtt{ccc}). \end{aligned}$$

Remember, we are no longer searching for the 495th letter of $\varphi^{31}(\mathtt{a})$ but rather the 434th letter of $\varphi^{29}(\mathtt{a})$.

FIND proceeds by determining which of the three cases to follow. GENERICLENGTH reports that $|X_1| \geq 434$ (in fact, $|X_1| = 434$) and hence the ith letter lies in X_1. We continue with Case 1 of the algorithm.

Since bdb grows polynomially rather than exponentially, we do a binary search to discover that $t = 0$, that is, the 434th letter of $\varphi^{27}(\mathtt{bdb})\varphi^{25}(\mathtt{bdb})\cdots\varphi(\mathtt{bdb})$, lies in the subword $\varphi(\mathtt{bdb})$. Thus, we have determined that the last a of the derivation path occurs when $n = (t+1)q+r = 3$. The value for i which yields the correct result is now $i - |\varphi^{29}(\mathtt{bdb})\cdots\varphi^{3}(\mathtt{bdb})| = 5$. We are now looking for the 5th letter of $\varphi^{3}(\mathtt{a})$.

We restart the algorithm with $n = 3$, $i = 5$, and $a = \mathtt{a}$, but after descending one level, we realize that $n \leq d$, and hence we call BASICTREEDESCENT returning the value d.

Case b: $i = 497$. The algorithm follows our first example closely until it determines which section of $\varphi^{29}(\mathtt{a})$, the ith letter falls in. The difference now is that $i = 436$ rather than 434. GENERICLENGTH calculates $|X_1| = 434$ and LENGTH reports that $|X_2| \geq i - |X_1|$.

Hence, the algorithm finishes with Case 2 by setting $n \leftarrow r = 1$, and $i \leftarrow i - |X_1| = 2$, and then calling BASICTREEDESCENT$(\varphi, 1, \mathtt{a}, 2)$ which returns the result e.

Case c: $i = 499$. Again, after descending two levels, we have $n = 29$, and $a = \mathtt{a}$, but now, $i = 438$. GENERICLENGTH and LENGTH tell us that $|X_1 X_2| = 437$. Hence the 438th letter of $\varphi^{29}(\mathtt{a})$ lies in X_3.

Regardless of whether ccc grows exponentially or not, Case 3 of our algorithm requires us to do a binary search among the factors of X_3. Our desired letter lies in the subword $\varphi(\mathtt{ccc})$ and hence $t = 0$.

We set $n \leftarrow (t+1)q + r = 3$, and $i \leftarrow i - |\varphi^{27}(\mathtt{bdb}) \cdots \varphi^{3}(\mathtt{bdb})| = 9$. Thus we have descended to the subtree of $\varphi^3(\mathtt{a})$. Since c grows exponentially, we are assured that n is not too large in relation to i. Accordingly, the algorithm invokes BASICTREEDESCENT which returns c.

4.3 Correctness and Analysis

We first prove the correctness of FIND, and then discuss its running time.

Theorem 15 FIND(φ, n, a, i) *returns the ith letter of* $\varphi^n(a)$.

Proof. We proceed by induction on n. When $n < d$, the algorithm calls BASICTREEDESCENT thus returning the correct answer. Otherwise, we assume that our algorithm works for values less than n. It remains to show that every time the value of n is reduced that a and i are modified in such a way that the ith letter of $\varphi^n(a)$ is the same.

The first part of the procedure descends the derivation tree one level at a time. Each time n is decreased by 1, the values for a, and i are z_t and $i - |\varphi^{n-1}(z_1 z_2 \cdots z_{t-1})|$ respectively, since the ith letter of $\varphi^n(a)$ lies in the subtree of z_t.

When a repeated letter a is found, we calculate the values for q, r, s, x and y such that for any $t < s$,

$$\varphi^n(a) = \varphi^{(s-1)q+r}(x) \cdots \varphi^{tq+r}(x)\varphi^{tq+r}(a)\varphi^{tq+r}(y) \cdots \varphi^{(s-1)q+r}(y).$$

Accordingly, for both Case 1 and Case 3, we set $i \leftarrow i - |\varphi^{(s-1)q+r}(x) \cdots \varphi^{(t+1)q+r}(x)|$, and $n \leftarrow (t+1)q + r$. Also, for Case 2, we set $n \leftarrow r$ and $i \leftarrow i - |\varphi^{(s-1)q+r}(x) \cdots \varphi^{(0)q+r}(x)|$.

Therefore, each time n is reduced, the values for a and i are also modified correctly. Hence FIND returns the ith letter of $\varphi^n(a)$. ∎

Theorems 7, 13 and 14 already give us the running times of the subroutines LENGTH GENERICLENGTH and BASICTREEDESCENT but we still need to show that whenever BASICTREEDESCENT is called by FIND, n is polynomial in $\log i$, w, or d. In our algorithm, we call BASICTREEDESCENT when $n \leq d$, when $n = r < q \leq d$, or when x or y grows exponentially. The following lemma shows that when x or y grows exponentially, then n is a polynomial in d and $\log i$.

Lemma 16 *Let $a \in \Sigma$, and $u, v \in \Sigma^*$ such that $\varphi^q(a) = uv$ for some $q < d$. If the ith letter of $\varphi^n(a)$ lies in the subword $\varphi^{n-q}(v)$, and u contains an exponentially growing letter b, then $n = O(d \log i)$.*

Proof. Recall from Section 2.4.3 that because u contains an exponentially growing letter, $|\varphi^n(u)|$ grows at least as fast as c^n where $c = \lfloor 2^{1/d} \rfloor$.

Since, $c^{n-q} \leq |\varphi^{n-q}(u)| < i$, then we know $(n-q)\log c < \log i$. Hence $n \leq (\log i)/(\log c) + q$. But $1/\log c = O(d)$ and $q < d$. Consequently, $n = O(d \log i)$. ∎

We apply Lemma 16 to Case 1 by setting $u = x$ and $v = ay$. For Case 3, we know a grows exponentially whenever y does, Hence, we apply Lemma 16 with $u = xa$ and $v = y$.

We have running times for the subroutines that FIND calls, but FIND also calls itself. The following lemma shows that FIND recursively calls itself at most d times by showing that each of the d letters can be the repeated letter of our algorithm at most one time.

Lemma 17 *Let q, r, and t be integers such that $r < q$, and $\varphi^q(a) = xay$ for some $x, y \in \Sigma^*$ and $a \in \Sigma$. If x grows polynomially and $i \leq |\varphi^{tq+r}(x)|$, or if y grows polynomially and $i > |\varphi^{tq+r}(xa)|$, then the derivation path p, of the ith letter of $\varphi^{(t+1)q+r}(a)$ contains the letter a exactly once at the root.*

Proof. Let p' denote the derivation path of the a occurring after the subword x at level q. Note that a only occurs in p' at the root and at level q.

Consider the case where x grows polynomially and $i \leq |\varphi^{tq+r}(x)|$. By level q, the path p takes an earlier branch than p' into the subword x. A second a can only occur in p after the path p branches from p'.

Thus, a second occurrence of a in p implies that a and consequently x are exponentially growing letters, a contradiction.

Otherwise y grows polynomially and $i > |\varphi^{tq+r}(xa)|$. Similarly, the path p descends into the subword y at level q. If p encounters an a after leaving the path p', then a grows exponentially, and y grows at least as fast as a, another contradiction. ∎

Finally, we state our main result: the running time of FIND is polynomial.

Theorem 18 *The number of bit operations used by* FIND *is*

$$O((d + \log i)w(\log n)d^4(d \log w + \log i)^2 + (\log n)^2 d^4(d^2 \log w + \log i)^2).$$

Proof. Let L be the running time of LENGTH, and $G\ell$, the running time of GENERICLENGTH.

In the worst case, the "for" loop of FIND executes d times, calculating the length of the words of subtrees w times, before one of the three cases is reached. Handling the cases takes at worst case $O(\log n G\ell L)$ time. This gives one instantiation of FIND a running time of

$$O(wdL + (\log n)G\ell).$$

We know FIND recursively calls itself at most d times before reaching the base case and calling BASICTREEDESCENT with $n < d \log i$. The running time of BASICTREEDESCENT then is

$$O(nwL) = O(wd(\log i)L).$$

Hence the total running time of FIND is

$$O(d(wdL + (\log n)G\ell + wd(\log i)L).$$

Substituting for $G\ell$ and L and rearranging gives us the final running time. ∎

The most significant difference between the running times of FIND and BASICTREEDESCENT is that there is only a $(\log n)^2$ term in the running time of FIND whereas there is a factor of n in the running time of BASICTREEDESCENT.

The upper bound we obtained in Theorem 18 overstates the actual running time obtained in practice. For most typical input data, FIND is rarely called more than once. It makes sense, then, to implement these procedures and analyze the actual running time of FIND for varying values of n, i, d, and w.

Table 4.1 reports the performance of FIND with $n = 1000$, and $n = 1000000$. The running times of BASICTREEDESCENT with $n = 1000$ are included for comparison. Each row of the table corresponds to a homomorphism with varying values of w and d and with alphabets of exponentially growing letters, polynomially growing letters, or both. For each homomorphism, we calculated the 100th, the 10000th, and the 1000000th letter of each word.

These results show a number of details concerning the performance of FIND on various inputs. The running times of both FIND and BASICTREEDESCENT have a high dependence on d. In practice however, d is not very large (typically less than 10 letters).

Notice that BASICTREEDESCENT does not perform as well as FIND does when $n = 1000$. For even larger n, the BASICTREEDESCENT would have an even worse performance compared with FIND. For example, by a linear extrapolation on n, obtaining the ith letter of $\varphi^{1000000}(a)$ using BASICTREEDESCENT would have taken about a day for each calculation. This shows that our new algorithm enables us to calculate letters of words produced by homomorphisms which were not obtainable before, particularly those produced with large values of n.

Table 4.1: Performance of FIND and BASICTREEDESCENT in seconds

			FIND $n = 1000$			FIND $n = 1000000$			BTD $n = 1000$		
Σ	d	w	i=100	10^4	10^6	i=100	10^4	10^6	i=100	10^4	10^6
exp	5	2	0.17	0.25	0.33	0.19	0.27	0.41	3.87	4.40	4.71
		10	0.47	0.75	1.17	0.42	0.59	0.86	9.90	10.35	10.65
	30	2	7.67	16.58	19.27	7.59	16.83	19.93	164.97	214.80	263.64
		10	7.24	10.89	14.13	7.19	10.78	14.09	151.58	200.73	211.40
poly	5	2	0.11	0.22	0.36	0.17	0.19	0.27	4.31	4.15	3.93
		10	0.26	0.30	0.53	0.40	0.41	0.46	9.75	18.19	42.05
	30	2	5.08	5.76	6.32	5.45	6.04	6.35	20.24	23.65	25.47
		10	4.79	4.96	6.27	5.33	5.62	5.68	59.30	116.15	76.08
mix	5	2	0.08	0.12	0.19	0.14	0.15	0.03	4.68	3.11	3.17
		10	0.18	0.23	0.26	0.24	0.33	0.34	7.82	8.34	7.54
	30	2	2.67	3.09	3.36	3.34	3.70	4.04	45.52	47.97	49.54
		10	6.67	9.39	11.98	6.78	9.53	12.10	151.26	161.09	159.70

4.3.1 Possible Improvements

We have shown that our algorithm for FIND takes

$$O((d + \log i)w(\log n)d^4(d \log w + \log i)^2 + (\log n)^2 d^4(d^2 \log w + \log i)^2)$$

steps. As we have said before, it is likely that this value is overstated. Specifically, that the exponent of d is 8 seems a little high. Although it would be nice to find the actual running time of FIND, our empirical results above, show that the running time is still very dependent upon the value of d. Perhaps changes could be made to improve the running time of FIND.

One possible idea is to use Salomaa's [Sal73] explicit calculation of the growth function $f_G(n)$ for the subroutine LENGTH and GENERICLENGTH. If such an approach were used, it would remain to show that the terms in the explicit polynomial do not get too large and hence not too expensive to compute.

Chapter 5

Implementation

The implementation of our algorithm mentioned above in Section 4.3 is accessible via the World Wide Web and is located at `http://www.math.uwaterloo.ca/~shallit/find/`. We chose to implement our algorithm in C++, although most other procedural languages would have also sufficed. If the reader wishes to implement the algorithm or perhaps even improve upon what we provide, this chapter provides details of our implementation. This chapter also serves as the documentation for our program.

5.1 Program Modules

The data types that our program deals with are not too complex. We want tools available to manipulate sets of letters, arbitrarily large integers, and matrices and vectors with such integers as elements. Our algorithm builds upon these to manage higher levels of abstraction such as homomorphisms and derivation trees. Here, we discuss each of the data types mentioned above as well as their associated methods.

A set of symbols is needed to represent things such as the letters accessible from `a`, or the set of mortal letters with respect to φ. The SymbolSet class stores this data as an array of sixty-two integers which correspond to the occurrences of each of the upper and lower case letters and ten numerals. The SymbolSet class has methods to add and delete letters to a set, to test whether a set contains a specified letter, to test whether two sets are equal or disjoint, to supply the Parikh vector or a string corresponding to a given set, and to supply the cardinality of the set itself.

With the intention of eventually replacing the data type used to store integers with a data type more capable of holding larger numbers, a separate module is used to define the LargeNumber data type. Initially, LargeNumbers were simply defined as doubles and, as we will see in Section 5.3 below, replaced with a data type capable of storing arbitrarily large numbers.

The code dealing with matrices and vectors is found in the MatrixUtil module. While there are many publicly available methods to implement matrices, we also require that our matrices have LargeNumbers as elements. Naturally, we use an array of LargeNumbers to store the data. In addition to the standard operations one might expect in a Matrix class (such as specifying the value of an element, or multiplying two matrices), we include the procedures POWER and SUPERPOWER as special methods.

Recall that we require POWER and SUPERPOWER to restrict each incidence matrix to only the letters accessible from a given word. Hence, we also include another method CLEAN to remove the rows and columns corresponding to these unaccessible letters. At first, CLEAN merely set the elements of each unnecessary row and column to zero, but too much time was wasted adding and multiplying zeros together. The method we finally arrived at stores two different values for the size of the array: one value to describe the number of elements allocated in memory, the other to specify the how many rows and columns are actually being used. Thus, the procedure CLEAN entails copying the useful elements of the matrix to the top left corner, and then reducing the number of rows and columns begin used.

Our most important data structure is our homomorphism. Paired with an alphabet, a homomorphism becomes a D0L-scheme and hence the DOLScheme is the most important class our algorithm deals with. The DOLScheme class has methods which allow us to create its incidence matrix and to calculate the various sets of symbols such as the polynomially growing letters. More importantly, however, the procedures LENGTH, BASICTREEDESCENT, GENERICLENGTH, and FIND are also part of the DOLScheme class. The DOLScheme stores the alphabet Σ as a SymbolSet and the homomorphism (implemented separately as the class Homom) is a linked list of character-string pairs.

As mentioned above, the DOLScheme module contains a substantial amount of our algorithm. The details of our algorithm have been explained and defined in Chapters 3 and 4 and our program is mostly a straightforward implementation of the given pseudocode. We say 'mostly' since the pseudocode does not provide tedious details such as those showing how to manage the allocation and deallocation of memory, to restrict a matrix to the letters accessible from a given word, or to implement a binary search. Since such details are not terribly enlightening, we omit them here.

In BASICTREEDESCENT it is not necessary to store any details of the derivation tree. In FIND, however, we need to keep track of the derivation path and the branches not taken at level j, that is, a_j, x_j, and y_j. The class DerivTree is written for such a purpose and contains several useful methods. For instance, we have methods to set and read the values of x_j, y_j, and a_j, to find whether a repeated letter has been encountered yet, and to return the set of letters contained in x and y (useful for determining whether x or y is exponentially growing, for example). Again, we simply use two arrays of SymbolSets and one array of letters to store this data.

While the Interface module and the Main module do not define any data structures or classes, they do provide a means to call the algorithm with a variety of inputs and parameters. The Interface module allows the user to input homomorphisms and arbitrarily large integers. The Main module determines whether the user should be prompted for input or not, whether BASICTREEDESCENT or FIND is called, and whether tracing should be turned on or off. Once all the necessary data has been input, then the ith letter of $\varphi^n(a)$ is calculated. Since both the Interface module and the Main module are not vital to our algorithm, they can be replaced by any other program that wishes to call FIND.

We finish this section with Table 5.1 to give a general idea of the size of our program and of each module.

Module	Lines of code
SymbolSet	145
LargeNumber	16
MatrixUtil	423
Homom	107
DOLScheme	460
DerivTree	93
Interface	192
Main	102
Total	1522

Table 5.1: The size of our program and its constituent modules

We note the number of lines in LargeNumber is fairly small, since it is just a redefinition of the types defined in the multiple precision package. The sizes of all the other modules, however, are proportional to their functionality.

5.2 Running the Program

Our program accepts input from standard input and outputs to standard output. Figure 5.1 illustrates the required input format. The homomorphism is specified with one production per line. Each line is of the form ax (spaces and tabs are ignored) where $\varphi(a) = x$. The `@` symbol signifies the end of φ's specification. Finally n, a, and i follow, delimited by white space.

Figure 5.2 gives a sample of our program's output. The output begins by printing the alphabet, the homomorphism, and the sets of vital letters, mortal letters, and so on. When

```
a bc
c da
b be
d dd
e e
@
100000
a
10000
```

Figure 5.1: Sample Input

FIND is called, it reports the values of n, a, and i, and then starts descending the derivation tree. In this case, the derivation path starts at `a`, and proceeds to `b` and then encounters another `b`; a repeated letter is found. We see that our algorithm enters Case 3 and does a binary search for $t = 9999$. Since the value of x is polynomial, the algorithm starts again with $n = 9999$, $a =$ `b` and $i = 10000$. This time we enter Case 2, BASICTREEDESCENT is called, and the result `e` is reported.

5.3 Multiple-precision Version

It is necessary to ensure that our algorithm can handle arbitrarily large integers. For instance, on the machine our algorithm was implemented on, the `double` data type does not have enough precision to calculate that $\varphi^{2^{100}+1}(b) = 1$ for $b \to b1$, $1 \to 10$, and $0 \to 00$. Hence we needed a package by providing a multiple-precision data type, capable of storing arbitrarily large numbers.

The requirements for our package are fairly simple. In addition to being able to assign the value of one integer a to another integer b, we required the ability to compute $a + b$, $a - b$, $a \times b$, $a \bmod b$, $\lfloor a/b \rfloor$, and to determine whether $a < b$, $a = b$, or $a > b$. Since most multiple precision packages already have this capability, I chose to use the Berkeley MP package which some Unix systems provide (in our case, SunOS version 4.1).

```
*************The Alphabet:
abcde
*************The Homomorphism:
e => e
d => dd
b => be
c => da
a => bc

Vital: abcde
Mortal:
Recursive: abcde
Expanding: d
Exponential: acd
Find:
  Find called with n=100000,a=a,i=10000.
  bb
Case 3
  -1<?<99998: 49999 24999 12499 6249 9374 10936 10155 9764 9959 10057
              10008 9983 9995 10001 9998 9999
  (poly) n=9999,a=b,i=10000.
  ee
Case 2
  Basic Tree Descent called with n=0,a=e,i=1.
  Answer, character number 10000 of Phi^100000(a): 'e'
```

Figure 5.2: Sample Output

There are many other multiple precision packages available if the reader wishes to use some other package. To do so, the LargeNumber module need only incorporate the new multiple precision package to provide the functions in Table 5.2.

move(a, b)	moves the value of a to b
madd(a, b, c)	puts the value $a + b$ into c
msub(a, b, c)	puts the value $a - b$ into c
mult(a, b, c)	puts the value $a * b$ into c
mdiv(a, b, c, d)	puts the value $\lfloor a/b \rfloor$ into c
	and the value a mod b into d
mcmp(a, b)	returns 0 if $a = b$, 1 if $a > b$, and -1 if $a < b$
mprint(a)	simply outputs the value of the number in decimal
itom(i)	returns a pointer to a new variable initialized to i
mfree(a)	frees the memory allocated for the variable a.

Table 5.2: Functions provided by LargeNumber

An unfortunate circumstance of Berkeley MP occurs when calculating expressions such as $j \leftarrow tq + r$. Such calculations must be broken up into the two statements: mult(t, q, j); madd(j, r, j);. which render the code a little less readable. In retrospect, the capabilities of C++ could have been used to define operators between a newly defined multiple-precision class — as opposed to just the data type it is now — which would have allowed for more complex expressions.

5.4 Testing for Correctness

If the reader plans on implementing our algorithm, or just modifying our implementation it may be useful to have the test cases we developed. Our original test set was meant to realize each case of our algorithm. As the program was being developed, various values of φ, n, a, and i would produce erroneous output. A new test case was added for each bug fixed until we finally arrived the test cases in Table 5.3.

φ	n	a	i	$\textsc{Find}(\varphi, n, a, i)$
a→b, b→c, c→dhe, d→f, e→g, f→b, g→b, h→ij, i →h, j→ ϵ	9	a	7	b
a→b, b→c, c→dhe, d→f, e→g, f→b, g→b, h→ij, i →h, j→ ϵ	30	a	199	c
a→bad, b→cb, c→c, d→dd	30	a	465	b
a→bad, b→cb, c→c, d→dd	30	a	466	a
a→bad, b→cb, c→c, d→dd	30	a	467	d
a→ab, b→cc, c→dd, d→ee, e→ ϵ	20000	a	6	d
a→A, A→aa	22	a	299	a
b→b1, 0→00, 1→10	200	b	$2^{100}+1$	1
a→bcde, b→c, c→d, d→e, e→a	68	a	58	a
a→cbab, b→cb, c→d, d→c, e→eabce	10000	e	1223	a
a→bbcbcecacdcddcdb, b→cdcbcded, c→dcde, d→ed, e→f, f→e	10	a	6940	d
a→bc, b→ac, c→ba	20	a	5	a
a→bc, b→be, c→da, d→dd, e→e	100000	a	10000	e

Table 5.3: Test input and the expected output

Chapter 6

Extensions

6.1 The Parikh Vector of a Prefix of $\varphi^n(a)$

Our algorithm is not only useful for calculating the ith letter of $\varphi^n(a)$. We can also use the same methods to obtain information about the *subwords* of $\varphi^n(a)$. Let p_i represent the prefix of $\varphi^n(a)$ of length i, and let $\mathbf{p}_i$ denote the Parikh vector of p_i. Here, we show how our algorithm is easily modified to calculate $\mathbf{p}_i$. Hence, by subtracting $\mathbf{p}_i$ from $\mathbf{p}_j$ where $i < j$, we also efficiently compute the letter distributions of the subword starting at position $i+1$ and ending at j.

The vector $\mathbf{p}_i$ is useful if, for example, the symbols of $\varphi^n(a)$ are the instructions of a turtle language. By using the total number of right turns and the total number of left turns (both values obtained from $\mathbf{p}_i$), we determine the orientation of the turtle after i instructions.

In Find and BasicTreeDescent, we treat i as a variable. To avoid confusion, let i be the original given value (hence, we want to compute $\mathbf{p}_i$), and let i' be the variable used and modified by our algorithm.

To compute $\mathbf{p}_i$, first note that at the end of Length, $|\varphi^n(a)|$ is calculated as $\mathbf{e}^T Z\mathbf{a}$. We modify Length to return, in addition to $|\varphi^n(a)|$, the Parikh vector of $\varphi^n(a)$ equivalent to $Z\mathbf{a}$. Similarly, we modify GenericLength to return the Parikh vector $Z\mathbf{x}$.

Secondly, i' is only decreased by values or sums of values calculated by the procedures LENGTH and GENERICLENGTH which correspond to the lengths of words preceding the ith letter of $\varphi^n(a)$. Now that GENERICLENGTH and LENGTH also return the Parikh vector of these words, we modify FIND and BASICTREEDESCENT to add these vectors to a running total $\mathbf{p}$ any time i' is decreased.

FIND always calls BASICTREEDESCENT as a base case, and BASICTREEDESCENT continues until $n = 0$. Because we have correctly chosen which branches to descend, we have $i' = 1$, when the calculation has completed. Therefore, $\mathbf{p} = \mathbf{p}_{i-1}$. To obtain $\mathbf{p}_i$, we simply add 1 to the appropriate element of $\mathbf{p}$ indicated by the final result of FIND.

To see that modifying our algorithm this way does not significantly affect the running time of the algorithm, we note that GENERICLENGTH and LENGTH were already calculating the Parikh vectors and then consequently summing their elements to obtain the results. The only additional calculations needed to compute $\mathbf{p}_i$ involve adding these same vectors to a running total $\mathbf{p}$.

6.2 Extending Homomorphisms to Transducers

It is natural to ask whether there exist polynomial-time algorithms for more general problems than the one we have dealt with thus far. Here we show that, by increasing the computing power of a homomorphism φ to that of a finite-state transducer, we significantly increase the difficulty of obtaining the ith letter of $\varphi^n(a)$. In fact, such computations require at least an exponential amount of time.

6.2.1 Finite-State Transducers

A *finite-state transducer*, or just *transducer*, is a machine $T = (Q, \Sigma, \delta, q_0, \Delta, \lambda)$ where Q is a finite set of states, Σ is the input alphabet, $\delta : (Q \times \Sigma) \to Q$ is the transition function, q_0 is the initial state, Δ is the output alphabet, and $\lambda : (Q \times \Sigma) \to \Delta^*$ is the output function.

Given input $x = x_1x_2x_3 \cdots x_n \in \Sigma^*$, a finite-state transducer starts at state q_0 and at the ith step, it enters the state $q_i = \delta(q_{i-1}, x_i)$ and appends $\lambda(q_{i-1}, x_i)$ to the output. Hence on input $x_1x_2x_3 \cdots x_n$, a finite-state transducer outputs

$$\lambda(q_0, x_1)\lambda(q_1, x_2) \cdots \lambda(q_{n-1}, x_n).$$

We define $\psi_T : \Sigma^* \to \Delta^*$ to be the function computed by the transducer T.

In a technical report [SS98] discussing the main results of this thesis, the following problem was suggested for further study.

Problem 19 (FIND$^{\text{TRANS}}$) *Let $\psi_T : \Sigma^* \to \Sigma^*$ be the function computed by the finite-state transducer $T = (Q, \Sigma, \delta, q_0, \Sigma, \lambda)$. Given integers $n \geq 0$ and $i \geq 1$ written in binary, and letters $a, b \in \Sigma$, does the ith letter of ${\psi_T}^n(a)$ equal b?*

The size of the input for FIND$^{\text{TRANS}}$ needs $O(\log n)$ and $O(\log i)$ bits to specify the integers n and i, respectively. Let d be the number of bits needed to specify T. We wish to discuss the deterministic time complexity of FIND$^{\text{TRANS}}$ as a function of the size of its input, that is, as a function of $(\log i)$, $(\log n)$, and d.

An efficient algorithm for FIND$^{\text{TRANS}}$ would be useful in studying words produced with the iterated application of a transducer. We say a transducer is *uniform* if $|\lambda(a, q)| = k$, *ϵ-free* if $|\lambda(a, q)| \geq 1$, and *growing* if $|\lambda(a, q)| \geq 2$ for all $a \in \Sigma, q \in Q$. Pansiot [Pan84] and Bleuzen [BG89] provide analyses of the fixed points of arbitrary transducers and their restricted versions (i.e., uniform, ϵ-free, and growing).

For example, consider the transducer

$$K = (Q, \Sigma, \delta, q_1, \Sigma, \lambda)$$

where $Q = \{q_1, q_2\}$, $\Sigma = \{1, 2\}$, $\delta(q_i, a) = q_{3-i}$, $\lambda(q_i, a) = i^a$ for all $q_i \in Q$ and $a \in \Sigma$. See

Figure 6.1 for a diagram. Note that a label a/b means "on input a, output b and enter the state pointed to by the arrow."

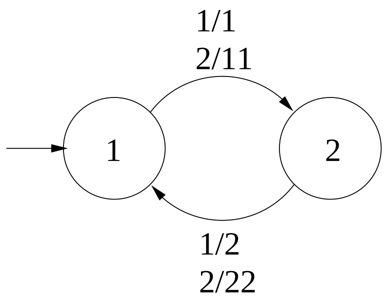

Figure 6.1: The transducer K.

The transducer K outputs a word whose sequence of run lengths is equivalent to the input word. Hence, the Kolakoski sequence

$$\mathbf{K} = 1221121221221121122\cdots,$$

(which has the property that the sequence of run lengths is equal to the sequence itself) is a fixed point of the transducer K. Despite a great deal of attention, the distribution of letters in $\mathbf{K}$ remains an open problem. For example, see [Kol65, Dek97, Ste96].

We revisit the example of interpreting a word as turtle instructions to produce the branching structure of a tree. Recall that calculating the ith letter of $\varphi^n(a)$ is analogous to calculating the *structure* of a twig without calculating the entire structure of the tree. Further, we can determine a twig's *orientation* by calculating the Parikh vector of the preceding letters and hence the total turns taken by the turtle before drawing the twig.

To determine the position of the twig relative to the initial position, the direction of the turtle for each forward move should be indicated somehow. By using a transducer, we can keep track of a finite number of directions and treat forward moves of varying directions distinctly. The calculation of a turtle's position after i instructions could then be obtained by calculating the distribution of forward moves for each distinct direction. (We note the similarity of this problem to the calculation of the distribution of letters in $\mathbf{K}$.)

In a final note, Pansiot [Pan84] mentions that arbitrary iterated transducers are "quite difficult to study since they allow the simulation of the computation of a Turing Machine". As we will see in Theorem 21, $\textsc{Find}^{\textsc{Trans}}$ requires at least an exponential amount of time; this result suggests an inherent difficulty in studying words such as $\mathbf{K}$.

6.2.2 The Turing Machine Model

The proof of Theorem 21 relies on the simulation of Turing machines. Of the many models of Turing machines, we will use the following, which is comparable to the model described by Papadimitriou [Pap94].

A *Turing machine* M is a quadruple (Q, Σ, δ, q_0), where Q is a finite set of states, Σ is a finite alphabet, $q_0 \in \Sigma$ is the starting state, and $\delta : (Q \times \Sigma) \to (Q \times \Sigma \times \{\leftarrow, -, \rightarrow\})$ is the transition function. If a Turing machine is in state $q_i \in Q$ with its tape head positioned over the letter $a_j \in \Sigma$, and if $\delta(q_i, a_j) = (q_k, a_\ell, \overset{?}{\leftrightarrow})$ where $q_k \in Q$, $a_\ell \in \Sigma$, and $\overset{?}{\leftrightarrow} \in \{\rightarrow, -, \leftarrow\}$ then in the next step, the Turing machine enters state q_k, overwrites a_j with a_ℓ, and then moves the tape head in the direction indicated by $\overset{?}{\leftrightarrow}$.

The input of a Turing machine is written on the first squares of an one-sided infinite tape. Further, a Turing machine always begins in state q_0 with its tape head positioned at the first square. For our purposes, we need only consider Turing machines that recognize recursive languages, that is, they either accept or reject every input. Accordingly, we require a Turing machine to complete its calculation by writing a "yes" symbol or a "no" symbol in the first square. Hence, we need not concern ourselves with Turing machines that move left past the end of the tape. The *language* of a Turing machine M is defined as

$$L(M) = \{x \in \Sigma^* : M \text{ accepts input } x\}.$$

We measure the *running time* of a Turing machine by counting the number of steps needed to accept or reject a given word.

6.2.3 Simulating a Turing Machine with a Transducer

We start with the definition of a specific word which describes the contents of the tape and the current position of the tape head. By manipulating such words, we will be able to simulate a Turing machine's computation with a transducer.

For any step of a Turing machine's calculation, let the *configuration word* be defined as

$$x_1 x_2 \cdots x_{i-1}(q_j, x_i)x_{i+1} \cdots x_m \triangleright,$$

where $x_1 x_2 \cdots x_m$ represents the contents of the tape (possibly containing a finite amount of trailing blanks), (q_j, x_i) indicates that the Turing machine is in state q_j and that the tape head is at position i, and the symbol $\triangleright$ signifies the infinite amount of blank tape to the right. For example, the word

$$(q_0, \mathtt{1})\ \mathtt{0}\ \mathtt{1}\ \mathtt{0}\ \mathtt{0}\ \mathtt{1}\ \triangleright$$

represents the first configuration of a Turing machine on input `101001`. These configuration words are closely related to the rows of the computation table described by Papadimitriou [Pap94, pg. 167].

The following lemma describes a finite-state transducer which maps one row of a computation table to the next, thus simulating one step of a Turing machine.

Lemma 20 *If M is a Turing machine, then there exists a finite-state transducer T, such that T maps any configuration word x to the configuration word which follows x in M's computation.*

Proof. The idea behind our proof is to delay the output of each square by one step, by using the states of the transducer. Thus, we are able to determine whether the delayed output should become an ordered pair, signifying the state and position of the Turing machine as a result of the tape head moving left. Once the tape head has been handled, the remainder of the transducer's calculation simply outputs the rest of the word as it receives it.

Consider an arbitrary Turing machine $M = (Q, \Sigma, \delta, q_0)$, where $Q = \{q_0, q_1, \ldots, q_{|Q|-1}\}$, $\Sigma = \{a_1, a_2, \ldots, a_{|\Sigma|}\}$, and $\delta : (Q \times \Sigma) \to (Q \times \Sigma \times \{\leftarrow, -, \rightarrow\})$. For convenience, we consider the blank symbol $\sqcup$ as part of Σ.

Then, let

$$T = (Q', \Sigma', \delta', A, \Sigma', \lambda'),$$

where

$$\Sigma' = \Sigma \cup (Q \times \Sigma) \cup \{\triangleright\},$$

$$Q' = \{A, B_1, \ldots, B_{|\Sigma|}, C_0, \ldots, C_{|Q|-1}, D\},$$

and both $\delta' : (Q' \times \Sigma') \to Q'$ and $\lambda' : (Q' \times \Sigma') \to \Sigma'^{*}$ are defined as in Table 6.1.

Current state	Input	$\delta(q_k, a_j)$	New state (δ')	Output (λ')
A	a_j	NA	B_j	ϵ
	(q_k, a_j)	$(q_\ell, a_m, -)$	D	(q_ℓ, a_m)
		$(q_\ell, a_m, \rightarrow)$	C_ℓ	a_m
B_i	a_j	NA	B_j	a_i
	(q_k, a_j)	$(q_\ell, a_m, \leftarrow)$	D	$(q_\ell, a_i)a_m$
		$(q_\ell, a_m, -)$	D	$a_i(q_\ell, a_m)$
		$(q_\ell, a_m, \rightarrow)$	C_ℓ	$a_i a_m$
C_i	a_j	NA	D	(q_i, a_j)
	$\triangleright$	NA	D	$(q_i, \sqcup)\triangleright$
D	a_j	NA	D	a_j
	$\triangleright$	NA	D	$\triangleright$
All other cases			Error : not legal	

Table 6.1: Definitions of δ' and λ'.

Intuitively, being in state A signifies that no scanning has occured yet and being in state B_i signifies that all but the previous square containing the symbol a_i has been output. Further, being in either state A or B_i means that the tape head has not been encountered yet.

Eventually, T encounters the tape head in either state A or in state B_i. If we are in state B_i, the previous square has not yet been output. Therefore, B_i is able to handle the case where the tape head moves left. At this point, enough information is known for the transducer to output the previous square as well as the current square (modified by the Turing machine in this step).

If the tape head moves left or stays still, no further work needs to be done. Hence, A and B_i send the transducer into state D which merely outputs the rest of the word as it is read. But in the case where the tape head postion must move right, A and B_i send the transducer to state C_j indicating that the next symbol read must become an ordered pair with the state q_j as the first item. Once the ordered pair is output by C_j, we enter state D and output the rest of the word as it is scanned.

We have not handled the case where the tape head moves past the left of the configuration word since this is an illegal Turing machine operation. On the other hand, what if the tape head were to move *right* into the trailing blanks indicated by $\triangleright$? Simply, when the state C_j encounters the symbol $\triangleright$, an extra blank is output as an ordered pair with the appropriate state as the first item. That is, the transducer outputs $(q_j, \sqcup)\ \triangleright$.

Hence the finite-state transducer T successfully maps a configuration word x to the configuration word of the step following x in the computation of M. ■

6.2.4 Classifying $\text{FIND}^{\text{TRANS}}$

We now discuss the main results of this section, namely, how $\text{FIND}^{\text{TRANS}}$ relates to the complexity classes of deterministic time. Formally, DTIME($f(n)$) refers to those languages which can be recognized by a deterministic Turing machine in time bounded by $f(n)$ and

$$\text{EXPTIME} = \bigcup_{k \geq 1} \text{DTIME}(2^{n^k}),$$

and

$$\text{2-EXPTIME} = \bigcup_{k \geq 1} \text{DTIME}(2^{2^{n^k}}).$$

For more on complexity classes, see [Pap94, GJ79].

Theorem 21 *The following two statements hold:*

(1) $\textsc{Find}^{\textsc{Trans}}$ *is in 2-EXPTIME.*

(2) $\textsc{Find}^{\textsc{Trans}}$ *is EXPTIME-hard.*

Proof. *(1)* $\textsc{Find}^{\textsc{Trans}}$ *is in 2-EXPTIME:* Let w be the maximum output for one symbol of input as in the case with homomorphisms. The brute-force solution to $\textsc{Find}^{\textsc{Trans}}$ applies the given transducer to a word n times and takes worst case $\sum_{i=0}^{n} w^i = O(nw^n)$ steps.

Let N be the size of the input to $\textsc{Find}^{\textsc{Trans}}$. Hence $nw^n \leq 2^N N^{2^N}$ since $\log n \leq N$ and $w \leq N$. Therefore the brute-force solution takes worst case $O(2^N N^{2^N})$ steps. Finally, we see $O(2^N N^{2^N}) \leq O(2^{2^{N^k}})$ by taking the log of each side:

$$N \log 2 + 2^N \log N \leq 2^{N^k} \log 2,$$

for all $k > 1$ and all sufficiently large N.

(2) $\textsc{Find}^{\textsc{Trans}}$ *is EXPTIME-hard:* We show this by reducing an EXPTIME-complete problem to $\textsc{Find}^{\textsc{Trans}}$. We use a *Karp reduction* (sometimes called *polynomial-time transformation*, or a *polynomial-time reduction*), since the class EXPTIME is closed under such reductions. Specifically, given an EXPTIME-complete language L, we must show there exists a function f computable in a polynomial amount of time such that $x \in L$ if and only if $f(x) \in \textsc{Find}^{\textsc{Trans}}$. See [GJ79] for further discussion on reductions and completeness.

Consider any EXPTIME-complete problem L, and let M_L be a Turing machine recognizing L in less than 2^{n^k} steps for some constant k. Further, let T_L be the transducer that simulates one step of M_L as described in the proof of Lemma 20.

Let x_0 be the configuration word of the first step of M_L on input x and let a_0 be some symbol not in Σ. The function $f(x)$ constructs a new transducer T from T_L by adding the symbol a_0 to the alphabet and having T output x_0 whenever it encounters an a_0. Augmenting T_L in this manner only requires time linear in $|x|$.

We want $f(x)$ to specify a value of n greater than the number of steps M_L needs to recognize x. Recall that the running time of M_L is less than $2^{|x|^k}$. Therefore, $f(x)$ specifies $n = 2^{|x|^k}$ in binary using a polynomial amount of time, since only $|x|^k$ bits are required.

The function $f(x)$ finishes by specifying $a = a_0$, $b =$ "yes", and $i = 1$.

It remains to show that $f(x) \in \textsc{Find}^{\textsc{Trans}}$ if and only if $x \in L$. By the construction of T and by Lemma 20, T simulates the calculation of M_L on input x. Since we have chosen n large enough, the ith letter (the first letter) of ${\psi_T}^n(a)$ contains the result of M_L: either a "yes" or a "no". Therefore ${\psi_T}^n(a) = b =$ "yes" if and only if M_L accepts x. ■

6.3 Extending D0L-systems to DIL-systems

In Section 2.2, we discussed other varieties of L-systems. We might ask, how efficiently can the ith letter of the nth word of some of these systems be computed? We show here that if we introduce interaction, then such calculations become significantly more difficult (as in the case with extending homomorphisms to transducers).

Recall from Section 2.2 that a D(k,ℓ)L-system is a deterministic L-system where the productions depend on context of size k to the left and of size ℓ to the right, and that DIL-systems are those D(k,ℓ)L-systems with $k+\ell > 0$. Let $G = (\Sigma, z, \theta)$ be a D(k,ℓ)L-system where θ is a context-sensitive set of replacement rules. Then we denote the nth word of the DIL-sequence G by $\theta^n(z)$. Here we consider the computational complexity of calculating the ith letter of $\theta^n(a)$ for $a \in \Sigma$ since we can add the production $a \to z$ whenever $|z| > 1$.

> **Problem 22** (FIND$^{\mathrm{DIL}}$) *Let $G = (\Sigma, a, \theta)$ be a D(k,ℓ)L-system where $k + \ell > 0$; let $b \in \Sigma$; and let n and i be integers. Does the ith letter of $\theta^n(a)$ equal b?*

Recall that any DIL-system is at least as powerful as a D1L-system. Hence, if there exists a D1L-system that simulates a Turing machine, then FIND$^{\mathrm{DIL}}$ is EXPTIME-hard by the same proof as Theorem 21.

Herman and van Dalen [Her69, van71] indepenently showed that a D1L-system can simulate a Turing machine in linear time. While this result is sufficient to show that FIND$^{\mathrm{DIL}}$ is EXPTIME-hard, Lemma 23, which is stated and proven below, explicitly provides the construction of a D1L-system which simulates an arbitrary Turing machine.

Before stating Lemma 23, we augment our definition of a configuration word by requiring a prime ($'$) be added to each square to the right of the tape head. Hence, for example, the first configuration of a Turing machine on input `101001` is now represented by the word $(q_0, \mathtt{1})\ \mathtt{0}'\ \mathtt{1}'\ \mathtt{0}'\ \mathtt{0}'\ \mathtt{1}'\ \triangleright$, and if $\delta(q_0, \mathtt{1}) = (q_7, \mathtt{0}, \rightarrow)$, then the second configuration word is $\mathtt{0}\ (q_7, \mathtt{0})\ \mathtt{1}'\ \mathtt{0}'\ \mathtt{0}'\ \mathtt{1}'\ \triangleright$.

Consider an arbitrary Turing machine $M = (Q, \Sigma, \delta, q_0)$ where $Q = \{q_0, \ldots, q_{|Q|-1}\}$, $\Sigma = \{a_1, \ldots, a_{|\Sigma|}\}$, and $\delta : (Q \times \Sigma) \rightarrow (Q \times \Sigma \times \{\leftarrow, -, \rightarrow\})$ as before. Further, let $\Sigma' = \{a' : a \in \Sigma\}$ and $\Delta = \Sigma \cup (Q \times \Sigma) \cup \Sigma' \cup \{\triangleright\}$.

Lemma 23 *If $xay\triangleright$ is an arbitrary configuration word where $x \in \Sigma^*$, $a \in (Q \times \Sigma)$, and $y \in \Sigma'^*$, then there exists a D$(1,0)$L-system $G = (\Delta, z, \theta)$ such that $\theta(xay\triangleright)$ is the configuration word following $xay\triangleright$ in M's computation.*

Proof. We define θ as in Table 6.2. Recall that for D$(1,0)$L-systems, if a letter a is replaced by a word x whenever it is preceded by a letter b, then we write $b\langle a\rangle \rightarrow x$.

Consider the case where $|x| > 0$ and $|y| > 0$. We write $x = x_0 x_1 \cdots x_n$ and $y = y_0 y_1 \cdots y_m$. Using productions (1) and (2), θ replaces x with $x_1 x_2 \cdots x_{n-1}$ and productions (5) and (6) replace $y_2 y_3 \cdots y_m\triangleright$ with itself. Further, we see that production (3) applies to a and (4)

$$(1) \qquad \epsilon \ \langle \ a_j \ \rangle \to \epsilon$$

$$(2) \qquad a_i \ \langle \ a_j \ \rangle \to a_i$$

$$(3) \qquad a_i \ \langle (q_k, a_j) \rangle \to \begin{cases} (q_\ell, a_i)\ a'_m, & \text{if } \delta(q_k, a_j) = (q_\ell, a_m, \leftarrow); \\ a_i\ (q_\ell, a_m), & \text{if } \delta(q_k, a_j) = (q_\ell, a_m, -); \\ a_i\ a_m, & \text{if } \delta(q_k, a_j) = (q_\ell, a_m, \rightarrow). \end{cases}$$

$$(4) \qquad (q_k, a_i) \ \langle \ a'_j \ \rangle \to \begin{cases} (q_\ell, a_j) & \text{if } \delta(q_k, a_i) = (q_\ell, a_m, \rightarrow); \\ a'_j & \text{otherwise.} \end{cases}$$

$$(5) \qquad a'_i \ \langle \ a'_j \ \rangle \to a'_j$$

$$(6) \qquad a'_i \ \langle \ \triangleright \ \rangle \to \triangleright$$

$$(7) \qquad \epsilon \ \langle (q_k, a_j) \rangle \to \begin{cases} (q_\ell, a_m) & \text{if } \delta(q_k, a_j) = (q_\ell, a_m, -); \\ a_m & \text{if } \delta(q_k, a_j) = (q_\ell, a_m, \rightarrow). \end{cases}$$

$$(8) \qquad (q_k, a_i) \ \langle \ \triangleright \ \rangle \to \begin{cases} (q_\ell, \sqcup)\ \triangleright & \text{if } \delta(q_k, a_i) = (q_\ell, a_m, \rightarrow); \\ \triangleright & \text{otherwise.} \end{cases}$$

Table 6.2: Definition of θ

applies to y_1. By considering the behaviour of θ when the tape head moves left, right or stays stationary, we can verify that the output is correct in each case.

If $|x| = 0$, then production (7) applies. If the tape head stays stationary or moves right, we see that (7) produces the tape's first square correctly. (We need not handle the case where the tape head moves left and consequently falls of the left end of the tape.)

If $|y| = 0$, then (8) applies to $\triangleright$ rather than (6). If the tape head stays stationary or moves left, then $\triangleright$ is simply replaced with itself. Otherwise, if the tape head should move right and into the trailing blanks, then a blank is appended by replacing $\triangleright$ with $(q_\ell, \sqcup)\ \triangleright$, where q_ℓ is the appropriate state. ■

Although we have shown that $\text{FIND}^{\text{TRANS}}$ and FIND^{DIL} are EXPTIME-hard, we may still want to find a class for which either of these problems is complete. There are some classes situated between 2-EXPTIME and EXPTIME, specifically,

$$\text{EXPTIME} \subseteq \text{NEXPTIME} \subseteq \text{EXPSPACE} \subseteq \text{2-EXPTIME}.$$

6.3.1 Other L-systems and Open Problems

We need not limit ourselves to studying FIND, $\text{FIND}^{\text{TRANS}}$, and FIND^{DIL} when there are other L-systems to consider; however defining FIND^{L} (where L represents the type of L-system in question) may not be as obvious for some types of L-systems. For instance, what exactly is the nth word of an extended L-system? (Recall that an extended L-system has non-terminals in its alphabet and omits any words containing a non-terminal.)

In an 0L-system, there is a *set* of words which can be obtained from the axiom after n nondeterministic replacements. Accordingly, we define the FIND^{0L} as the following: given letters $a, b \in \Sigma$, integers i and n, and a nondeterministic homomorphism φ, does there exist a word $x \in \varphi^n(a)$ such that the ith letter of $x = b$? I conjecture that FIND^{0L} is in NP (for no other reason than that it is a nondeterministic version of our main problem which has been shown to be in P). Similarly, we can introduce nondeterminism to DIL-systems and ask whether FIND^{IL} is NEXPTIME-complete.

Tabled systems are another form of nondeterminism. We define a similar problem to the one described above for 0L-systems as $\text{FIND}^{\text{DT0L}}$: given $a, b \in \Sigma$, integers i and n and a table of homomorphsisms Φ, does there exist a word $x = \varphi_0\varphi_1 \cdots \varphi_n(a)$ where $\varphi_k \in \Phi$, such that the ith letter of x equals b?

We might also ask whether there are propagating L-systems which constrain the L-system in question enough to allow more efficient algorithms? For example, although FIND^{DIL} is EXPTIME-complete, is there is a polynomial-time algorithm for $\text{FIND}^{\text{PDIL}}$?

We finish by formalizing the open problems suggested above, and by mentioning a few others.

Problem 24 *For which computational complexity classes, if any, are the following problems* complete*:* FIND*,* $\text{FIND}^{\text{TRANS}}$*,* FIND^{DIL}*,* FIND^{0L}*,* FIND^{IL}*,* $\text{FIND}^{\text{DT0L}}$*,* $\text{FIND}^{\text{PD0L}}$*, and* $\text{FIND}^{\text{PDIL}}$*?*

Problem 25 *What is the running time of the* optimal *algorithm for* FIND*?*

Let $\text{FIND}^{\text{TRANS}(\epsilon\text{-FREE})}$, $\text{FIND}^{\text{TRANS}(\text{GROWING})}$, and $\text{FIND}^{\text{TRANS}(\text{UNIFORM})}$, be defined as $\text{FIND}^{\text{TRANS}}$ with the appropriate restrictions on the transducer T. We can now ask the following:

Problem 26 *For which computational complexity classes, if any, are the following problems complete:* $\text{FIND}^{\text{TRANS}(\epsilon\text{-FREE})}$, $\text{FIND}^{\text{TRANS}(\text{GROWING})}$, *and* $\text{FIND}^{\text{TRANS}(\text{UNIFORM})}$ *?*

$\text{FIND}^{\text{TRANS}}$ is EXPTIME-hard, but this problem deals with *arbitrary* transducers. There may be more efficient algorithms to calculate the ith letter of strings produced by *specific* transducers.

M.-w. Wang [personal communication] discovered a deterministic algorithm to compute the ith letter of the Kolakoski sequence in $O(i)$ steps, using $O(\log i)$ space. Since the size of the input is $N = \log i$, the calculation is in EXPTIME and in DSPACE(N). With this in mind, our open problem is:

Problem 27 *What is the running time of the optimal algorithm to compute the ith letter of the Kolakoski sequence?*

$|x|$ length of the word x, 3

$|x|_a$ number of occurrences of the letter a in the word x, 4

0L-system a non-deterministic version of a D0L-system, 12

2-EXPTIME equal to DTIME($2^{2^{n^k}}$), 72

a, b, c letters in Σ, 3

`a,b,c,...,A,B,C,...,1,2,3...` actual values of letters in Σ (in an example), 1

$\mathbf{a}, \mathbf{p}_i, \mathbf{x}, \mathbf{y}, \mathbf{z}$ the Parikh vectors of a, p_i, x, y, and z respectively, 19

a_j the letter encountered after descending j levels down the derivation tree, 44

accessible the letter b is accessible from a when $\varphi^n(a) = x_1 b x_2$ for some $n \geq 0$ and $x_1, x_2 \in \Sigma^*$, 10

axiom the initial word of an L-system, 6

configuration word a word describing the configuration of a Turing machine at some step during its computation, 69

d the size of φ's domain, 3

D0L equivalence problem given D0L-systems G_1 and G_2, does $L(G_1) = L(G_2)$?, 17

D0L membership problem given a word x and a D0L-system G, is $x \in L(G)$?, 16

D0L realization problem does there exist a D0L-system which satisfies a given growth function?, 23

D0L-languages the set of words in a D0L-sequence; equal to $\{\varphi^n(z) \;:\; n \geq 0\}$, 15

D0L-sequence the sequence $z, \varphi(z), \varphi^2(z), \ldots$ produced by a D0L-system, 2

D0L-system deterministic L-system without interactions, 6

D1L-system a D(0,1)L-system or a D(1,0)L-system, 14

D2L-system a D(1,1)L-system, 14

Δ the output alphabet of a finite-state transducer, 65

δ a transition function, 65

depth of a homomorphism size of a homomorphism's domain; equal to $|\Sigma|$, 3

depth of a locally catenative D0L-sequence the maximum of $i_1, \ldots, i_k$ (as in the definition of locally catenative), 10

derivation path a path which leads from the root of a derivation tree to some node of the tree, 11

derivation tree an ordered tree where every non-leaf node b has children labeled with the letters of $\varphi(b)$, 11

DIL-system any D(k,ℓ)L-system where $k + \ell > 0$, 14

D(k,ℓ)**L-system** a context-sensitive L-system; k and l specify the left and right context respectively, 13

DT0L-system a tabled D0L-system, 12

DTIME$(f(n))$ the set of languages which can be recognized in $O(f(n))$ time, 71

e column vector whose entries are all ones, 20

ED0L-system an extended D0L-system, 13

ϵ the empty string, 1

ϵ**-free** any transducer where $|\lambda(a,q)| \geq 1$, for all $a \in \Sigma, q \in Q$, 66

expanding any letter a such that $\varphi^n(a) = x_1ax_2ax_3$ for some integer $n > 0$ and some $x_1, x_2, x_3 \in \Sigma^*$, 11

exponential growth any growth function f_G where $f_G(n) \geq c^n$ for some $c > 1$, 21

EXPTIME equal to DTIME(2^{n^k}), 72

$f_G(n)$ the growth function of a D0L-system G, 19

FIND$^{\mathrm{X}}$ the calculation of the ith letter of the nth word of an L-system of type X, 76

finite-state transducer an automaton which implements some function $\Sigma^* \to \Delta^*$, 65

fixed point of f any word x such that $f(x) = x$ where f is a homomorphism (or transducer), 24

G an L-system, 6

growing any transducer where $|\lambda(a,q)| \geq 2$, for all $a \in \Sigma, q \in Q$, 66

growth function the function describing the word lengths of a D0L-sequence; equal to $|\varphi^n(z)|$, 19

homomorphism a mapping (often φ) which preserves concatenation; that is $\varphi(xy) = \varphi(x)\varphi(y)$, 3

I the identity matrix, 20

i the index of the letter we wish to compute, 3

Im a function which deletes the mortal letters from a word, 32

immortal any letter that is not mortal, 10

incidence matrix a matrix whose entries are the letter frequencies in the productions of a homomorphism, 19

K the Kolakoski sequence; equal to $122112122\cdots$, 67

Karp reduction we say "A reduces to B" when $x \in A$ iff $f(x) \in B$, where f is computable in polynomial time, 72

L-system see Lindenmayer system, 6

λ the output function of a finite-state transducer, 66

limited growth any growth function f_G where $0 < f_G(n) < m$ for some constant m, 21

Lindenmayer system a system of production rules developed to model the growth of biological organisms, 6

locally catenative D0L-sequence contains words comprised *entirely* of earlier words in the sequence, 9

$M(\varphi)$ the incidence matrix of φ; equal to $(|\varphi(b)|_a)_{a,b\in\Sigma}$, 19

malignant growth exponential growth, 22

mono-recursive any letter a such that $\varphi^n(a) = x_1 a x_2$ for some $n > 0$, a not accessible from $x_1 x_2$, 11

mortal any letter a such that $\varphi^n(a) = \epsilon$ for some $n > 0$, 10

n number of times we wish to compose φ with itself, 3

normal growth polynomial growth, 22

O the zero matrix, 35

p a running total in the computation of $\mathbf{p}_i$, 65

p_i the prefix of $\varphi^n(a)$ of length i, 64

Parikh vector a vector describing the letter distribution of a word x; equal to $(|x|_a)_{a\in\Sigma}$, 4

PD0L-system a propagating D0L-system, 13

φ a homomorphism, 3

φ^j a homomorphism composed with itself j times, 3

$\pi_k(L(G))$ the subword complexity of $L(G)$, 18

polynomial growth any growth function f_G where $p(n) \geq f_G(n)$ for some non-constant polynomial $p(n)$, 21

ψ_T the function computed by the finite-state transducer T, 66

Q a finite set of states, 65

q_0 an initial state, 65

rank the maximum exponent of a letter a's (polynomial) growth function, 22

recursive any letter a such that a is accessible from itself, 10

Σ finite alphabet, 3

Σ^* the finite words over Σ, 3

Σ^ω the one-sided infinite words over Σ, 24

$\Sigma^{\mathbb{Z}}$ the two-sided infinite words over Σ, 24

subword complexity the number of distinct subwords of length k in all the words in a language, 18

terminating growth any growth function f_G where $f_G(n) = 0$ for some value of n, 21

θ a context-sensitive set of replacement rules in a DIL-system, 73

transducer see finite-state transducer, 65

Turing machine any quadruple which describes the machine model that we have chosen, 68

turtle language a programming language with a set of commands to a "turtle" to move around a plane and leave a trail, 6

uniform any transducer where $|\lambda(a, q)| = k$, for some constant k, and for all $a \in \Sigma, q \in Q$, 66

$\mathbf{v}$ a column vector with 1's indicating the immortal letters of Σ and 0's indicating the mortal letters, 30

vital see immortal, 10

w the width of φ, 3

width of a homomorphism the maximum $|\varphi(a)|$ over all $a \in \Sigma$, 3

x, y words in Σ^*, 3

X_1, X_2, X_3 three factors of $\varphi^n(a)$; specified only when a repeated letter is found, 46

x_j, y_j words of the jth step of a derivation such that $\varphi(a_{j-1}) = x_j a_j y_j$, 44

z the axiom of an L-system, 6

Bibliography

[AD90] J.P. Allouche and F. Dress. Tours de Hanoï et automates. *Theoretical Informatics and Applications*, 24(1):1–15, 1990.

[AS98] J.P. Allouche and J. Shallit. Automatic Sequences: Theory, Applications, Generalizations. In preparation, 1998.

[Ben82] L. Benson. *The Cacti of the United States and Canada.* Stanford University Press, 1982.

[BG89] N. Bleuzen-Guernalec. On a possible classification of real-time constructed sequences. *Theoretical Computer Science*, 65:143–148, 1989.

[Blo85] J. Bloomenthal. Modelling the mighty maple. In *SIGGRAPH '85*, volume 19, pages 305–311, 1985.

[BS96] E. Bach and J. Shallit. *Algorithmic Number Theory.* MIT Press, 1996.

[CF77] K. Culik II and I. Fris. The decidability of the equivalence problem for D0L-systems. *Information and Control*, 35:20–39, 1977.

[Cho90] C. Choffrut. Iterated substitutions and locally catenative systems: a decidability result in the binary case. In *Automata, Languages and Programming*, volume 700 of *Lecture Notes in Computer Science*, pages 490–500. Springer–Verlag, 1990.

[Cho92] C. Choffrut. Iterated substitutions and locally catenative systems: a decidability result in the binary case. In G. Rozenberg and A. Salomaa, editors, *Lindenmayer Systems: Impacts on Theoretical Computer Science, Computer Graphics, and Developmental Biology*, pages 49–92. Springer–Verlag, 1992.

[Cul78] K. Culik II. The ultimate equivalence problem for D0L systems. *Acta Informatica*, 10(1):79–84, 1978.

[Dek97] F. M. Dekking. What is the long range order in the Kolakoski sequence? In R. V. Moody, editor, *The Mathematics of Long-Range Aperiodic Order*, volume 489 of *NATO Adv. Sci. Inst. Ser. C Math. Phys. Sci.*, pages 115–125. Kluwer, 1997.

[DHN85] S. Demko, L. Hodges, and B. Naylor. Construction of fractal objects with iterated function systems. In *SIGGRAPH '85*, volume 19, pages 271–278, 1985.

[Dou72] P. G. Doucet. On the membership question in some Lindenmayer systems. *Indag. Math.*, 34:45–52, 1972.

[Dou73] P. G. Doucet. The growth of word length in D0L-systems. In *Open House in Unusual Automata Theory (Aarhus University, Technical Report DAIMI PB-15)*, pages 83–94, 1973.

[DR71] P. G. Doucet and G. Rozenberg. On 0L-languages. *Information and Control*, 19:302–318, 1971.

[ELR75] A. Ehrenfeucht, K. P. Lee, and G. Rozenberg. Subword complexities of various classes of deterministic developmental languages without interaction. *Theoretical Computer Science*, 1:59–75, 1975.

[ER73] A. Ehrenfeucht and G. Rozenberg. A limit theorem for sets of subwords in deterministic T0L languages. *Information Processing Letters*, 2:70–73, 1973.

[ER74] A. Ehrenfeucht and G. Rozenberg. D0L systems with rank. In G. Rozenberg and A. Salomaa, editors, *L Systems*, volume 15 of *Lecture Notes in Computer Science*, pages 136–141. Springer – Verlag, 1974.

[ER78] A. Ehrenfeucht and G. Rozenberg. Simplification of homomorphisms. *Information and Control*, 38:298–309, 1978.

[ER79] J. Engelfriet and G. Rozenberg. Equality languages and fixed point languages. *Information and Control*, 43:20–49, 1979.

[Fri88] M. Friendly. *Advanced Logo: A Language for Learning*. Lawrence Erlbaum Associates, 1988.

[Gar67] M. Gardner. Mathematical games. *Scientific America*, 216(3):124–129, March 1967. Additional material in **216** (4) (April 1967), 116–123; **217** (1) (July 1967), 112–116.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability*. Freeman, 1979.

[Gre83] J. J. Grefenstette. Stability in L systems. *Theoretical Computer Science*, 24:53–71, 1983.

[Har77] T. Harju. A polynomial recognition algorithm for the EDT0L languages. *Elektro. Inform. Kybernetik.*, 13:169–177, 1977.

[Hea81] T. Head. Fixed languages and the adult languages of $0L$ schemes. *Internat. J. Comput. Math.*, 10:103–107, 1981.

[Her69] G. T. Herman. The computing ability of a developmental model for filamentous organisms. *Journal of Theoretical Biology*, 25, 1969.

[HL86] T. Head and B. Lando. Fixed and stationary ω-words and ω-languages. In G. Rozenberg and A. Salomaa, editors, *The Book of L*, pages 147–156. Springer–Verlag, 1986.

[HS98] D. Hamm and J. Shallit. Characterization of finite and one-sided infinite fixed points of morphisms on free monoids. *submitted*, 1998.

[HU69] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison–Wesley, 1969.

[HV76] G. T. Herman and P. Vitányi. Growth functions associated with biological development. *American Mathematical Monthly*, 83:1–15, 1976.

[HW75] G. T. Herman and A. Walker. Context free languages in biological systems. *Intern. J. Computer Math.*, 4:369–391, 1975.

[JS76] N. Jones and S. Skyum. Recognition of deterministic ET0L languages in polynomial time. Technical Report DAIMI PB-63, University of Aarhus, 1976.

[JS77a] N. Jones and S. Skyum. Lower bounds on the complexity of some problems concerning L systems. Technical Report DAIMI PB-70, University of Aarhus, 1977.

[JS77b] N. Jones and S. Skyum. Recognition of deterministic ET0L languages in logarithmic space. *Information and Control*, 35:175–181, 1977.

[JS77c] N. Jones and S. Skyum. Upper bounds on the complexity of some problems concerning L systems. Technical Report DAIMI PB-69, University of Aarhus, 1977.

[JS79] N. Jones and S. Skyum. Complexity of some problems concerning L systems. *Math. Systems Theory*, 13:29–43, 1979.

[JS81] N. Jones and S. Skyum. A note on the complexity of general D0L membership. *SIAM J. Comput.*, 10:114–117, 1981.

[Knu69] D. Knuth. *The Art of Computer Programming*, volume 2. Addison–Wesley, 1969.

[Kol65] W. Kolakoski. Elementary problem 5304. *American Mathematical Monthly*, 72:681–682, 1965. Solution in **73**.

[KS79] Y. Kobuchi and S. Seki. Decision problems of locally catenative property for DIL-systems. *Information and Control*, 43:266–279, 1979.

[Lin77] M. Linna. The decidability of the D0L prefix problem. *Intern. J. Computer Math*, 6:127–142, 1977.

[LJ92] A. Lindenmayer and H. Jürgensen. Grammars of development: discrete-state models for growth, differentiation, and gene expression in modular organisms. In *Lindenmayer Systems*, pages 3–21. Springer–Verlag, 1992.

[LR73] A. Lindenmayer and G. Rozenberg. Developmental systems with locally catenative formulas. *Acta Informatica*, 2:214–248, 1973.

[LR74] K. P. Lee and G. Rozenberg. The length sets of D0L languages are uniformly bounded. *Information Processing Letters*, 2:185–188, 1974.

[Pan84] J.-J. Pansiot. On various classes of infinite words obtained by iterated mappings. In *Automata on Infinte Words*, volume 192 of *Lecture Notes in Computer Science*, pages 188–197. Springer–Verlag, 1984.

[Pap94] C. H. Papadimitriou. *Computational Complexity.* Addison–Wesley, 1994.

[PH89] P. Prusinkiewicz and J. Hanan. *Lindenmayer Systems, Fractals, and Plants*, volume 79 of *Lecture Notes in Biomathematics.* Springer–Verlag, 1989.

[PL90] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants.* Springer – Verlag, 1990.

[PS73] A. Paz and A. Salomaa. Integral sequential word functions and growth equivalence of Lindenmayer systems. *Information and Control*, 23:313–343, 1973.

[Roz74] G. Rozenberg. D0L sequences. *Discrete Mathematics*, 7:324–347, 1974.

[Sal73] A. Salomaa. On exponential growth in Lindenmayer systems. *Koninkl. Nederl. Akademie Van Wetenschappen Ser. A*, 76:23–29, 1973.

[Sal75a] A. Salomaa. Comparative decision problems between sequential and parallel rewriting. In *Proc. of the Intern. Symposium on Uniformly Structured Automata and Logic*, pages 62–66, 1975.

[Sal75b] A. Salomaa. Formal power series and growth functions of Lindenmayer systems. In *Mathematical Foundations of Computer Science, 1975*, volume 32 of *Lecture Notes in Computer Science*, pages 143–154. Springer–Verlag, 1975.

[San98] C. Sansom. The born loser. *The Kitchener-Waterloo Record*, page C5, November 5 1998. Comic strip.

[Sha98] J. Shallit. Number theory and formal languages. In D. A. Hejhal, J. Friedman, M. C. Gutzwiller, and A. M. Odlyzko, editors, *Proc. IMA Conference on Emerging Applications of Number Theory*, volume 109 of *IMA Volumes in Mathematics and Applications*, 1998. To appear.

[SQ79] A. L. Szilard. and R. E. Quinton. An interpretation for D0L systems by computer graphics. *The Science Terrapin*, 4:8–13, 1979.

[SS98] J. Shallit and D. Swart. An efficient algorithm for computing the ith digit of $\varphi^n(a)$. Technical Report CS-98-21, University of Waterloo, 1998.

[Ste96] R. Steacy. Structure in the Kolakoski sequence. *Bull. European Assoc. Theor. Comput. Sci.*, 59:173–182, June 1996.

[Sud77] I. H. Sudborough. The time and tape complexity of developmental languages. In *Automata, Languages and Programming*, volume 52 of *Lecture Notes in Computer Science*, pages 509–523. Springer–Verlag, 1977.

[SW98] J. Shallit and M.-w. Wang. On two-sided infinite fixed points of morphisms. Manuscript, submitted, 1998.

[Szi71] A. L. Szilard. Growth functions of Lindenmayer systems. Technical Report 4, University of Western Ontario, Dept. of Comp. Sci., 1971.

[Thi79] G. Thierrin. Context-free and stationary languages. *Intern. J. Computer Math*, 7:297–301, 1979.

[van71] D. van Dalen. A note on some systems of Lindenmayer. *Mathematical Systems Theory*, 5:128–140, 1971.

[Vit72] P. Vitányi. D0L languages and a feasible solution for a word problem. Technical Report MR 138/72, Mathematisch Centrum, 1972.

[Vit73] P. Vitányi. Structure of growth in Lindenmayer systems. *Koninkl. Nederl. Akademie Van Wetenschappen Ser. A*, 76:248–253, 1973.

[Vit74] P. Vitányi. On the size of D0L languages. In G. Rozenberg and A. Salomaa, editors, *L Systems*, volume 15 of *Lecture Notes in Computer Science*, pages 78–92. Springer – Verlag, 1974.

[Vit76] P. Vitányi. Digraphs associated with D0L-systems. In A. Lindenmayer and G. Rozenberg, editors, *Automata, Languages and Development*, pages 335–346. North-Holland Publishing Company, 1976.

[Vit80] P. Vitányi. *Lindenmayer systems: structure, languages, and growth functions*, volume 96. Mathematical Centre Tracts, Amsterdam, 1980.