A Generalization of Square-free Strings

A GENERALIZATION OF SQUARE-FREE STRINGS

BY

NEERJA MHASKAR, B.Tech., M.S.

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTING & SOFTWARE

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Doctor of Philosophy (2016) McMaster University

(Dept. of Computing & Software) Hamilton, Ontario, Canada

TITLE:	A Generalization of Square-free Strings
AUTHOR:	Neerja Mhaskar B.Tech., (Mechanical Engineering) Jawaharlal Nehru Technological University Hyderabad, India M.S., (Engineering Science) Louisiana State University, Baton Rouge, USA
SUPERVISOR:	Professor Michael Soltys
CO-SUPERVISOR:	Professor Ryszard Janicki
NUMBER OF PAGES:	xiii, 116

To my family

Abstract

Our research is in the general area of String Algorithms and Combinatorics on Words. Specifically, we study a generalization of square-free strings, shuffle properties of strings, and formalizing the reasoning about finite strings.

The existence of infinitely long square-free strings (strings with no adjacent repeating word blocks) over a three (or more) letter finite set (referred to as Alphabet) is a well-established result. A natural generalization of this problem is that only subsets of the alphabet with predefined cardinality are available, while selecting symbols of the square-free string. This problem has been studied by several authors, and the lowest possible bound on the cardinality of the subset given is four. The problem remains open for subset size three and we investigate this question. We show that square-free strings exist in several specialized cases of the problem and propose approaches to solve the problem, ranging from patterns in strings to Proof Complexity. We also study the shuffle property (analogous to shuffling a deck of cards labeled with symbols) of strings, and explore the relationship between string shuffle and graphs, and show that large classes of graphs can be represented with special type of strings.

Finally, we propose a theory of strings, that formalizes the reasoning about finite strings. By engaging in this line of research, we hope to bring the richness of the advanced field of Proof Complexity to Stringology.

Acknowledgments

I would like to express my sincere gratitude to my advisor Professor Michael Soltys for his continuous guidance and support over the years. I would particularly like to thank him for giving me the opportunity and freedom to work on the problems that interested me. Without his encouragement, patience, and positivity, I would not have been able to complete my thesis and in particular enjoy my doctoral studies. He is truly the best advisor and mentor to have.

I would like to thank my co-supervisor Professor Ryszard Janicki for his support over the last couple of years. In particular for the financial support during my last academic term.

I would like to thank the other members of my supervisory committee, Professor Douglas Down, Professor Bill Smyth and Professor Maxime Crochemore for reviewing the current work and their valuable comments. It was a particular honor to have Maxime Crochemore as the external examiner for this thesis.

Finally, I would like to thank my family: my parents for their encouragement, my husband for his support, patience, and faith in me, and my kids for their unconditional love and patience throughout this study.

Notations and Abbreviations

The following notation is used throughout the thesis.

Table 1: Notations and Abbreviations List

Symbol	**Description**
i, j, k, l, m, n	Positive Integers
$\boldsymbol{X}_1, \boldsymbol{X}_2, \ldots, \boldsymbol{X}_n$	String Variables (Variables ranging over Strings)
X, Y, Z	Variables
Σ	Finite Alphabet/Finite set of symbols
$a, b, c, d, a_1, a_2, \ldots, a_n$	Alphabet symbols
$[\boldsymbol{a}\text{-}\boldsymbol{z}]$	Strings/Words
$\boldsymbol{w}[1..n]$	String $\boldsymbol{w}$ represented in array form.
$\boldsymbol{w}_1\boldsymbol{w}_2\ldots\boldsymbol{w}_n$	String $\boldsymbol{w}$ represented as sequence of symbols $\boldsymbol{w}_i$, $1 \leq i \leq n$
$\lvert\boldsymbol{w}\rvert$	Length of string $\boldsymbol{w}$
ε	Empty string (String of length zero)
Σ^*	Set of all finite strings over Σ
Σ^+	$\Sigma^* - \varepsilon$
Σ_k	Fixed generic alphabet of k symbols
$\Sigma_{\boldsymbol{w}}$	Set of symbols occurring in $\boldsymbol{w}$

Table 1 – *Notation contd...*

Symbol	**Description**
$\|\boldsymbol{w}\|_a$	Frequency of a symbol 'a' in string $\boldsymbol{w}$
$\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n$	List of strings $\boldsymbol{x}_1$ to $\boldsymbol{x}_n$
$\boldsymbol{u} \cdot \boldsymbol{v}$	String $\boldsymbol{u}$ concatenated with string $\boldsymbol{v}$
f, g, h	Functions/Morphisms
$L = L_1, L_2, \ldots, L_n$	Alphabet list with n alphabets
$\|L\|$	Length of alphabet list L
Σ_L	$L_1 \cup L_2 \cup \ldots \cup L_n$ (Symbols of an alphabet list L)
$\hat{L}$	List L in normalized form
$\mathcal{L}$	Class of lists with a certain property
$\mathcal{L}_k$	Class of lists $L = L_1, L_2, \ldots, L_n$, s.t $\|L_i\| = k$
$\mathcal{L}_{\Sigma_k}$	Class of lists $L = L_1, L_2, \ldots, L_n$, s.t $L_i = [k]$
β	Border of a string or Logical Formula
$[\mathcal{A} - \mathcal{Z}]$	String Patterns or Sequences
$\mathcal{C}(n)$	n-th Offending Suffix
$\mathcal{C}_s(n)$	n-th Shortest Offending Suffix
$\mathcal{Z}_n$	n-th Zimin word
CMP	Consistent Mapping Problem
shuffle	Shuffle Problem
parity	Parity Problem
$\boldsymbol{w} = \boldsymbol{u} \odot \boldsymbol{v}$	$\boldsymbol{w}$ is a shuffle of $\boldsymbol{u}, \boldsymbol{v}$
Shuffle($\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{w}$)	Shuffle Predicate
Parity($\boldsymbol{x}$)	Parity Predicate

Table 1 – *Notation contd...*

Symbol	**Description**
Shuffle	Language of Shuffle
Parity	Language of Parity
$\Pi_{i=1}^{n} \boldsymbol{x}_i$	$\boldsymbol{x}_1 \cdot \boldsymbol{x}_2 \cdot \ldots \cdot \boldsymbol{x}_n$, where $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n$ are strings
$\mathcal{S}$	Three sorted logical theory for strings

Contents

List of Figures

Chapter 1

Introduction

The Study of Combinatorics on Words has seen great interest in recent years. This study which is an area of discrete mathematics, primarily deals with strings, which are ordered sequences of symbols from a finite set referred to as Alphabet. It dates back to early nineteen hundreds and the pioneering work done by Axel Thue. In 1906 and 1912, Thue published his now classical papers [Thue, 1906] and [Thue, 1912], which are primarily about repetitions in strings over a fixed finite alphabet. Berstel in [Berstel, 1995], gives an excellent translation of the work done in these papers.

Since Thue's original work was published in an obscure journal, it was noticed much later and became "classical" only in the later part of the twentieth century. As a result, some of his work was rediscovered over and over again, for example see [Aršon, 1937], [Morse and Hedlund, 1944], and [Leech, 1957]. The major growth of interest in this area was only after fifty years, since Thue published his first paper. [Berstel and Perrin, 2007], gives a nice overview of the origin of this study. A lot of research has been done in this area since, and has applications in a variety of fields ranging from abstract algebra, bioinformatics, formal languages to the genetic code.

While in Combinatorics on Words, we generally study properties of infinitely long strings, in the area of String Algorithms, we study properties of strings of finite length. Examples of such properties are: pattern matching and studying the maximum number of squares in a given string of finite length. For a comprehensive overview of these subjects see: [Lothaire, 1983], [Lothaire, 2002], [Lothaire, 2005], [Allouche and Shallit, 2003], [Berstel et al., 2008], [Karhumäki, 2004], [Smyth, 2003] and [Crochemore and Rytter, 1994].

1.1 Motivation

Our research is primarily on strings over a finite alphabet. In the area of avoiding repetitions in strings, it is motivated by the following open problem (posed in [Grytczuk et al., 2013]): can we construct a square-free string (string which does not contain the subword $\boldsymbol{vv}$ for any word $\boldsymbol{v}$) of length n, over an ordered list of (finite) alphabets, $L = L_1, L_2, \ldots, L_n$, where each alphabet has exactly three symbols and the i-th symbol of the square-free string is chosen from L_i, the i-th alphabet? This problem is a generalization of square-free strings and the classical results in the area are not applicable to this setting, which makes the problem even more intriguing.

Although much progress has been done in understanding shuffle, many questions regarding shuffle remain open. For example, does shuffle square (given a string $\boldsymbol{w}$, is it a shuffle of some $\boldsymbol{x}$ with itself), remain **NP**-hard for some alphabets with fewer than seven symbols? Motivated by this, we explore further properties of shuffle in order to better understand this operation and as a result learn an interesting relationship between graphs and string shuffle.

Finally, while working on finite strings, we observed that although many techniques have been developed over the years in the area of String Algorithms to prove properties of finite strings, there is no unifying theory or framework formalizing it. We propose a unifying theory of strings based on a three sorted logical theory, which we call $\mathcal{S}$. By engaging in this line of research, we hope to bring the richness of the advanced field of Proof Complexity to Stringology, and eventually create a unifying theory of strings.

1.2 Contribution

This thesis is the result of four published/accepted papers listed under "List of co-authored publications" by Neerja Mhaskar and Michael Soltys. Here we give a summary of our published research and other important results yet to be published. The main results of our work from [Mhaskar and Soltys, 2015b] and [Mhaskar and Soltys, 2016] are listed below:

- In Theorem 9 (Theorem 1, [Mhaskar and Soltys, 2015b]), we show that alphabet lists having any one of the properties: SDR, union, consistent mapping, and partition are all admissible, that is, a square-free string exists over such lists. In the proofs, we also give constructions of square-free strings over such lists.

- In Lemma 7 (Lemma 6, [Mhaskar and Soltys, 2015b]), we show that given an alphabet list $L = L_1, L_2, \ldots, L_n$, where $|L_i| = 3$, finding if it has a consistent mapping, i.e., the consistent mapping problem (CMP) is **NP**-hard. We show this by reducing the 3-colorability of planar graphs to CMP.

- We give an alternate proof of Thue's result in Section 2.2.

- In Lemma 2, we show that if a string $\boldsymbol{w}$ (constructed using Algorithm 1, [Grytczuk et al., 2013]) is square-free, then for any symbol a, either $\boldsymbol{w'} = \boldsymbol{w}a$ is still square-free, or $\boldsymbol{w'}$ has a unique square (consisting of a suffix of $\boldsymbol{w'}$). This was assumed but not proved in [Grytczuk et al., 2013].

- In Chapter 5 on future directions, we present an Offending Suffix pattern (Section 5.1.1) and later relate it to forcing squares in strings over an alphabet list in Lemma 26 (Theorem 1, [Mhaskar and Soltys, 2016]), and in Lemma 30 (Lemma 4, [Mhaskar and Soltys, 2016]) we give a characterization of square-free strings using borders.

In our paper [Mhaskar and Soltys, 2015c], we study the shuffle property of strings. We show that many string operations can be expressed with string shuffle. We also explore the relationship between graphs and string shuffle, and show that many classes of graphs can be represented with strings exhibiting shuffle properties. Particularly, in Lemma 7 of the paper we show that: if a graph or its complement is isomorphic to a transitive closure of a tree then it can be represented with a string exhibiting shuffle property.

Finally in our paper [Mhaskar and Soltys, 2015a], for the first time (as far as we know) we give a basic logical theory, denoted as $\mathcal{S}$, for finite strings and state various string constructs as formulas in the theory. In Theorem 1, one of the primary results of the paper, we show that, if we prove the existence of a string $\boldsymbol{u}$ with some given property in $\mathcal{S}_1$, then we can construct such a string with a polytime algorithm.

1.3 Thesis Outline

In Chapter 1, we introduce and define the terminology used in the thesis.

In Chapter 2, we give the background on work done in the area of a particular generalization of square-free strings, and show that solutions exist in many specialized cases of the open problem. Some of the results in this Chapter can be found in [Mhaskar and Soltys, 2015b].

In Chapter 3, we study the shuffle property of strings and explore an interesting relationship between strings exhibiting shuffle properties and graphs. Most of the contents of this chapter can be found in our paper [Mhaskar and Soltys, 2015c].

In Chapter 4, we give a formalism for Stringology consisting of a three sorted logic theory designed to capture reasoning about finite strings. The contents of this chapter can be found in our paper [Mhaskar and Soltys, 2015a].

In Chapter 5, we conclude the thesis by proposing various strategies for solving the open problems in each area, and give a summary list of open problems. Some of the results in this Chapter can be found in [Mhaskar and Soltys, 2016].

1.4 Definitions

Strings (Words)

An *alphabet* is a set of symbols, and Σ is usually used to represent a finite *alphabet*. The elements of an alphabet are referred to as *symbols* (or *letters*). In this thesis, we assume $|\Sigma| \neq 0$. A *string* (or a *word*) over Σ, is an ordered sequence of symbols from it. Formally, $\boldsymbol{w} = \boldsymbol{w}_1 \boldsymbol{w}_2 \dots \boldsymbol{w}_n$, where for each i, $\boldsymbol{w}_i \in \Sigma$, is a symbol. For example,

if $\Sigma = \{a, b, c\}$, then the string *abacabaabbabacacc* is a string over Σ. In order to emphasize the array structure of $\boldsymbol{w}$, we sometimes represent it as $\boldsymbol{w}[1..n]$. The *length* of a string $\boldsymbol{w}$ is denoted by $|\boldsymbol{w}|$. The set of all finite length strings over Σ is denoted by Σ^*. The empty string is denoted by ε, and it is the string of length zero. The set of all finite strings over Σ not containing ε is denoted by Σ^+. We denote Σ_k to be a fixed alphabet of k symbols, $\Sigma_{\boldsymbol{w}}$ to be the set of symbols occurring in the string $\boldsymbol{w}$, that is, $\Sigma_{\boldsymbol{w}} = \{\boldsymbol{w}_i \in \Sigma : 1 \leq i \leq k \wedge \boldsymbol{w} = \boldsymbol{w}_1\boldsymbol{w}_2 \ldots \boldsymbol{w}_k\}$, and $|\boldsymbol{w}|_a$ to be the number of times the symbol a occurs in $\boldsymbol{w}$.

A string $\boldsymbol{v}$ is a *subword* (also known as a *substring* or a *factor*) of $\boldsymbol{w}$, if $\boldsymbol{v} = \boldsymbol{w}_i\boldsymbol{w}_{i+1} \ldots \boldsymbol{w}_j$, where $1 \leq i \leq j \leq n$. If $i = 1$, then $\boldsymbol{v}$ is a *prefix* of $\boldsymbol{w}$ and if $j < n$, $\boldsymbol{v}$ is a *proper prefix* of $\boldsymbol{w}$. If $j = n$, then $\boldsymbol{v}$ is a *suffix* of $\boldsymbol{w}$ and if $i > 1$, then $\boldsymbol{v}$ is a *proper suffix* of $\boldsymbol{w}$. We can express that $\boldsymbol{v}$ is a subword more succinctly using array representation as $\boldsymbol{v} = \boldsymbol{w}[i..j]$. A word $\boldsymbol{v}$ is a *subsequence* of a string $\boldsymbol{w}$ if the symbols of $\boldsymbol{v}$ appear in $\boldsymbol{v}$ in the same order as they appear in $\boldsymbol{w}$. Note that the symbols of $\boldsymbol{v}$ do not necessarily appear contiguously in $\boldsymbol{w}$. Hence, any subword is a subsequence, but the reverse is not true. For example, if $\boldsymbol{w} = ababbacaaacbaba$ is a string, then '*ababbac*' is a proper prefix, '*acaaacbaba*' is a proper suffix, '*abbacaaacba*' is a subword and '*abbca*' is a subsequence of $\boldsymbol{w}$.

We define string concatenation (using the '$\cdot$' operator) in the standard way as follows: If $\boldsymbol{u} = \boldsymbol{u}_1\boldsymbol{u}_2 \ldots \boldsymbol{u}_m$ and $\boldsymbol{v} = \boldsymbol{v}_1\boldsymbol{v}_2 \ldots \boldsymbol{v}_n$ are two strings then $\boldsymbol{u} \cdot \boldsymbol{v}$ (read as $\boldsymbol{u}$ concatenated with $\boldsymbol{v}$) is a new string obtained by juxtaposing $\boldsymbol{u}$ and $\boldsymbol{v}$. Therefore $\boldsymbol{u} \cdot \boldsymbol{v} = \boldsymbol{u}_1\boldsymbol{u}_2 \ldots \boldsymbol{u}_m\boldsymbol{v}_1\boldsymbol{v}_2 \ldots \boldsymbol{v}_n$. For example, if $\boldsymbol{u} = aaaaa$ and $\boldsymbol{v} = bbbbbb$, then $\boldsymbol{u} \cdot \boldsymbol{v} = aaaaabbbbbb$.

Morphisms

A map $h : \Sigma^* \to \Delta^*$, where Σ and Δ are finite alphabets, is called a morphism if for all $x, y \in \Sigma^*$, $h(\boldsymbol{xy}) = h(\boldsymbol{x})h(\boldsymbol{y})$. A morphism is said to be *non-erasing* if for all $\boldsymbol{w} \in \Sigma^*$, $|h(\boldsymbol{w})| \geq |\boldsymbol{w}|$. An example of a morphism is given in (1.1). A morphism h, over some alphabet Σ, is called square-free if $h(\boldsymbol{w})$ is square-free for every square-free word $\boldsymbol{w}$ over Σ. The morphism given in (1.1) is also a square-free morphism. Towards the end of Section 2.2, we discuss a couple of existing characterizations of square-free morphisms which give an easy way to check if a morphism is square-free.

Repetitions

A string $\boldsymbol{w}$ has a *square* (or a *repetition*) if there exists a word $\boldsymbol{v}$ such that $\boldsymbol{vv}$ is a subword of $\boldsymbol{w}$. We say that $\boldsymbol{w}$ is *square-free* (or *non-repetitive*) if no such subword exists. For example, the word "*repe***titi***on*" has a square "**titi**" while the word "*square*" has no repetition.

It is easy to see that a square-free word of length four or more does not exist over a binary alphabet. However, Thue in his seminal paper [Thue, 1906] proved Theorem 1[1], and showed the existence of square-free strings over a three letter alphabet $\Sigma = \{1, 2, 3\}$ through the use of the following morphism[2].

[1]Since Thue's original paper is in German, and various sources state his result in different ways, we were not able to quote verbatim the Theorem stated in his original paper.

[2]Usually, Thue's result is over $\Sigma = \{a, b, c\}$, but we use $\Sigma = \{1, 2, 3\}$ given in [Grytczuk et al., 2013].

$$g = \begin{cases} 1 \mapsto 12312 \\ 2 \mapsto 131232 \\ 3 \mapsto 1323132 \end{cases} \tag{1.1}$$

Given a string $\boldsymbol{w} \in \Sigma_3^*$, $g(\boldsymbol{w})$ is the string with every symbol of $\boldsymbol{w}$ replaced by its corresponding mapping in the morphism g. Formally, $g(\boldsymbol{w}) = g(\boldsymbol{w}_1\boldsymbol{w}_2 \dots \boldsymbol{w}_n) = g(\boldsymbol{w}_1)g(\boldsymbol{w}_2) \dots g(\boldsymbol{w}_n)$. Note that the substitutions are simultaneous. Therefore, by repeatedly applying the morphism given in (1.1) to any square-free string over Σ_3 of non-zero length, one can construct an arbitrarily long square-free string. For example, using $\boldsymbol{w} = 132123$, a square-free string, and by applying the morphism g to it we get:

$$g(\boldsymbol{w}) = g(1)g(3)g(2)g(1)g(2)g(3) = 123121323132132123212312131232132 3132,$$

which is also square free.

Theorem 1 (A. Thue)**.** *If $\boldsymbol{w} \in \Sigma_3^*$ is a square-free string, then so is $g(\boldsymbol{w})$.*

In Section 2.2, we give an alternate proof of Theorem 1. For a detailed discussion on the topic see [Berstel et al., 2008], [Smyth, 2003].

Squares are the basic type of repetitions in strings, and we primarily study avoiding this type of repetition in strings. Some of the other types of repetitions include cubes (words with three adjacent repeating blocks), overlaps (words of the form $a\boldsymbol{v}a\boldsymbol{v}a$, where $a \in \Sigma$ and $\boldsymbol{v} \in \Sigma^*$), and abelian squares (words of the form $\boldsymbol{w}\boldsymbol{w}'$, where $|\boldsymbol{w}| = |\boldsymbol{w}'|$ and $\boldsymbol{w}'$ is a permutation of $\boldsymbol{w}$) etc. To further study other types of repetitions, see for example, [Shallit, 2009], [Rampersad, 2007].

Words over Alphabet Lists

An *alphabet list* is an ordered list of finite subsets called alphabets consisting of symbols, where all the alphabets have the same cardinality. Let $L = L_1, L_2, \ldots, L_n$, be an ordered list of alphabets. A string $\boldsymbol{w}$ is said to be *over* a list L, if $\boldsymbol{w} = \boldsymbol{w}_1\boldsymbol{w}_2\ldots\boldsymbol{w}_n$ where for all i, $\boldsymbol{w}_i \in L_i$. Note that there are no conditions imposed on the alphabets L_i's: they may be equal, disjoint, or have elements in common. The only condition on $\boldsymbol{w}$ is that the i-th symbol of $\boldsymbol{w}$ must be selected from the i-th alphabet of L, i.e., $\boldsymbol{w}_i \in L_i$. The alphabet set for the list $L = L_1, L_2, \ldots, L_n$ is denoted by $\Sigma_L = L_1 \cup L_2 \cup \cdots \cup L_n$. L^+ denotes the set of strings over the list L. For example, if $L = \{\{a, b, c\}, \{c, d, e\}, \{a, 1, 2\}\}$, then $ac1 \in L^+$ but $2ca \notin L^+$.

Given a square-free string $\boldsymbol{w}$ over a list $L = L_1, L_2, \ldots, L_n$, where $L \in \mathcal{L}_3$, we say that the alphabet L_{n+1} *forces a repetition on* $\boldsymbol{w}$ if for all $a \in L_{n+1}$, $\boldsymbol{w}a$ has a repetition. For example, if $L = \{\{a, b, c\}, \{a, b, c\}, \{a, b, c\}, \{a, b, c\}, \{a, b, c\}, \{a, b, c\}, \{a, b, c\}\}$ and $\boldsymbol{w} = abacaba$, then the alphabet $\{a, b, c\}$ forces a repetition on $\boldsymbol{w}$, as the strings $\boldsymbol{w}a$, $\boldsymbol{w}b$ and $\boldsymbol{w}c$ each has a repetition.

Admissibility

We introduce the concept of admissibility of lists. We say that an alphabet list L is *admissible* if L^+ contains a square-free string. For example, the alphabet list $L = \{\{a, b, c\}, \{1, 2, 3\}, \{a, c, 2\}, \{b, 3, c\}\}$, is admissible as the string '$a1c3$' over L is square-free.

Let $\mathcal{L}$ represent a *class* of lists; the intention is for $\mathcal{L}$ to denote lists with a given property. For example, we are going to use $\mathcal{L}_{\Sigma_k}$ to denote the class of lists

$L = L_1, L_2, \ldots, L_n$, where for each $i \in [n] = \{1, 2, \ldots, n\}$, $L_i = \Sigma_k$, and $\mathcal{L}_k$ will denote the class of all lists $L = L_1, L_2, \ldots, L_n$, where for each $i \in [n]$, $|L_i| = k$, that is, those lists consisting of alphabets of size k. Note that $\mathcal{L}_{\Sigma_k} \subseteq \mathcal{L}_k$. We say that a class of list $\mathcal{L}$ is admissible if *every* list $L \in \mathcal{L}$ is admissible. An example of an admissible class of lists is the class $\mathcal{L}_{\Sigma_3}$ (Thue's result).

Borders

A *border* β of a string $\boldsymbol{w}$, is a subword that is both a proper prefix and proper suffix of $\boldsymbol{w}$. Note that the proper prefix and proper suffix may overlap as shown in Figure 1.1. A word can have more than one border. For example, if $\boldsymbol{w} = 121345121$, then it has a border 1 and 121 and the empty border ε. See [Smyth, 2003] for a detailed study on borders.

Period

A period of a string $\boldsymbol{w}$ is an integer p, such that $\forall i \in \{1, 2, \ldots, |\boldsymbol{w}| - p\} \boldsymbol{w}_i = \boldsymbol{w}_{i+p}$ and $0 < p \leq |\boldsymbol{w}|$. A string can have several periods, but we are mostly interested in the shortest period of the string. Also, every string has a period equal to its length. Observe the duality between borders and periods. A string $\boldsymbol{w}$ has a period p if and only if it has a border of length $|\boldsymbol{w}| - p$. In the definition of period, i ranges over $\{1, 2, \ldots, |\boldsymbol{w}| - p\}$, where $|\boldsymbol{w}| - p$ is the length of a border of $\boldsymbol{w}$. See [Smyth, 2003] and [Crochemore and Rytter, 1994] for a detailed discussion on periods.

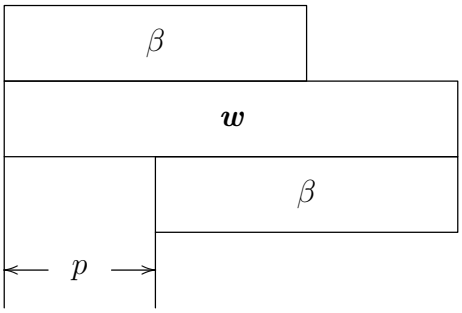

Figure 1.1: Border β of length $|\boldsymbol{w}| - p$, and period p of string $\boldsymbol{w}$

Pattern

Given an alphabet Σ, $\Delta = \{\boldsymbol{X}_1, \boldsymbol{X}_2, \boldsymbol{X}_3, \ldots, a_1, a_2, a_3, \ldots\}$ is a set of variables, where the variables $\boldsymbol{X}_i$ range over Σ^*, and the variables a_i range over Σ, that is unit length strings. A *pattern* is a non empty string over Δ^*; for example, $\mathcal{P} = \boldsymbol{X}_1 a_1 \boldsymbol{X}_1$ is a pattern representing all strings where the first half equals the second half, and the two halves are separated by a single symbol. Intuitively, patterns are "templates" for strings. Note that some authors define patterns as being words over variables with no restriction on the size of the variables (see [Berstel and Perrin, 2007]), but we find the definition given here as more amenable to our purpose.

We say that a word $\boldsymbol{w}$ over some alphabet Σ *conforms* to a pattern $\mathcal{P}$ if there is a morphism $h : \Delta^* \longrightarrow \Sigma^*$, such that $h(\mathcal{P}) = \boldsymbol{w}$.

Avoidable and Unavoidable Patterns

We say that a pattern is *avoidable*, if strings of arbitrary length exist, such that no subword of the string conforms to the pattern, otherwise it is said to be *unavoidable*. For example, the pattern $\boldsymbol{XX}$ is unavoidable for all strings in $\{\boldsymbol{w} \in \{0,1\}^* : |\boldsymbol{w}| \geq 4\}$, but it is avoidable over the ternary alphabet (Thue's result restated in Theorem 1).

Zimin Words

The idea of unavoidable patterns was developed independently by Bean et al. in [Bean et al., 1979] and by Zimin in [Zimin, 1982]. *Zimin words* (also known as *sesquipowers*) constitute a certain class of unavoidable patterns. The n-th Zimin word, $\mathcal{Z}_n$, is defined recursively over the alphabet $\Delta = \{\boldsymbol{X}_1, \boldsymbol{X}_2, \ldots, \boldsymbol{X}_n\}$ of variables of type string as follows:

$$\begin{aligned} \mathcal{Z}_1 &= \boldsymbol{X}_1, \text{and for } n > 1, \\ \mathcal{Z}_n &= \mathcal{Z}_{n-1}\boldsymbol{X}_n\mathcal{Z}_{n-1}. \end{aligned} \tag{1.2}$$

The understanding is that the variables $\boldsymbol{X}_i$, where $0 \leq i \leq n$, range over words over some fixed background alphabet Σ. In [Zimin, 1982], Zimin proved that the n-th Zimin word, $\mathcal{Z}_n$, constitutes an unavoidable pattern for strings over Σ_n. For example, the pattern $\mathcal{Z}_3 = \boldsymbol{X}_1\boldsymbol{X}_2\boldsymbol{X}_1\boldsymbol{X}_3\boldsymbol{X}_1\boldsymbol{X}_2\boldsymbol{X}_1$ is unavoidable over a three letter alphabet Σ_3, that is, there always exists a non-erasing morphism h such that $h(\mathcal{Z}_3)$ is a subword of $\boldsymbol{w}$, where $\boldsymbol{w}$ is a sufficiently long word over Σ_3. It turns out that for all strings $\boldsymbol{w} \in \Sigma_3^*$, such that $|\boldsymbol{w}| \geq 29$, the pattern $\mathcal{Z}_3$ is unavoidable. See [Cooper and Rorabaugh, 2014] for bounds on Zimin word avoidance. For details on Zimin patterns, see [Zimin, 1982], [Berstel et al., 2008], [Rampersad and Shallit, 2012], [Berstel and Perrin, 2007], and [Cooper and Rorabaugh, 2014].

Graphs

An *undirected graph* $G = (V, E)$, consists of a nonempty set V of vertices (or nodes), and a set of edges E. A graph in which each edge connects two different vertices is called a *simple graph*. All graphs discussed in the thesis are simple graphs.

A *directed graph* G, is a graph consisting of directed edges or arcs, and each directed edge is associated with an ordered pair of vertices.

A *bipartite graph* $G = (V, E)$, is a graph whose node set can be partitioned into sets X and Y in such a way that every edge in G has one end in X and the other end in Y.

The *transitive closure* of a graph $G = (V, E)$, is the graph $G' = (V, E')$, such that, E' is the smallest superset of E with the following property: if $(v, w), (w, u)$ are two edges in E', then $(v, u) \in E'$.

The *complement of a graph* $G = (V, E)$, denoted as $G^c = (V, E^c)$ is a graph obtained from G such that, $e \in E$ if and only if $e \notin E^c$.

Parity

The function parity is defined on binary strings. It returns 1 if the number of ones in the string is odd, otherwise it returns 0. That is, $\mathsf{parity}(\boldsymbol{u}) = \sum_{i=1}^{|\boldsymbol{u}|} \boldsymbol{u}_i \mod 2$, where $\boldsymbol{u}$ is a binary string. The predicate Parity($\boldsymbol{u}$) of a binary string $\boldsymbol{u}$ holds, if the number of ones in the string is odd.

Shuffle

If $\boldsymbol{u}$, $\boldsymbol{v}$, and $\boldsymbol{w}$ are strings over an alphabet Σ, then $\boldsymbol{w}$ is said to be a *shuffle* of $\boldsymbol{u}$ and $\boldsymbol{v}$ provided there are (possibly empty) strings $\boldsymbol{u}_i$ and $\boldsymbol{v}_i$ such that $\boldsymbol{u} = \boldsymbol{u}_1\boldsymbol{u}_2\cdots\boldsymbol{u}_k$ and $\boldsymbol{v} = \boldsymbol{v}_1\boldsymbol{v}_2\cdots\boldsymbol{v}_k$ and $\boldsymbol{w} = \boldsymbol{u}_1\boldsymbol{v}_1\boldsymbol{u}_2\boldsymbol{v}_2\cdots\boldsymbol{u}_k\boldsymbol{v}_k$. We use $\boldsymbol{w} = \boldsymbol{u} \odot \boldsymbol{v}$, to denote that $\boldsymbol{w}$ is a shuffle of $\boldsymbol{u}$ and $\boldsymbol{v}$. A necessary condition for the existence of a shuffle is that, the length of $\boldsymbol{w}$ must be equal to the sum of the lengths of $\boldsymbol{u}$ and $\boldsymbol{v}$. For example, if $\boldsymbol{u} = \mathbf{100110101}$ and $\boldsymbol{v} = 010011010$, then $\boldsymbol{w} = \mathbf{100}0\mathbf{11}00\mathbf{1}\mathbf{0}\mathbf{1}\mathbf{0}\mathbf{1}1010 = \boldsymbol{u} \odot \boldsymbol{v}$. Also

note that some $\boldsymbol{u}_i$ and $\boldsymbol{v}_j$ may be ε. For example, if $\boldsymbol{u}_1 \neq \varepsilon$, $\boldsymbol{u}_2 \neq \varepsilon$, and $\boldsymbol{v}_1 = \varepsilon$, then this would mean that we take the first two symbols of $\boldsymbol{u}$ before we take any symbols from $\boldsymbol{v}$. In short, by choosing certain $\boldsymbol{u}_i$'s and $\boldsymbol{v}_j$'s equal to ε's, we can obtain any shuffle from an ostensibly strictly alternating shuffle.

Shuffle can be defined over any alphabet, but we work with the binary alphabet $\Sigma = \{0, 1\}$. The predicate Shuffle$(\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w})$ holds if $\boldsymbol{w}$ is a shuffle of $\boldsymbol{u}$ and $\boldsymbol{v}$, that is,

$$\text{Shuffle}(\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}) \Leftrightarrow \boldsymbol{w} = \boldsymbol{u} \odot \boldsymbol{v}$$

We use the following naming convention: "shuffle" and "parity" denote the generic problems, and Shuffle$(\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w})$, Parity$(\boldsymbol{u})$ denote the corresponding predicates, and Shuffle and Parity without arguments denote the corresponding languages.

Pair-string

A *pair-string* is a string where each symbol occurs exactly twice, i.e., where $\forall \boldsymbol{a} \in \Sigma_{\boldsymbol{w}}, |\boldsymbol{w}|_a = 2$. A pair-string can also be viewed as a shuffle of some string $\boldsymbol{u}$ with some permutation of the symbols of $\boldsymbol{u}$, where every symbol of $\boldsymbol{u}$ is distinct. For any pair-string $\boldsymbol{w}$, if a is a symbol in $\boldsymbol{w}$, then we denote the first occurrence of a as a^1 and the second occurrence of a as a^2.

Pair-string with Monge/Anti-Monge Property

A graph G, is said to have the *anti-Monge* property if the edges in the graph are crossed or non-nested, see Figure 1.2. If the edges in the graph are nested, then it is said to have the *Monge* property, see Figure 1.3.

Figure 1.2: Example of non-nested edges in graph.

Figure 1.3: Example of nested edges in graph.

A *bipartite graph* $G_{\boldsymbol{w}}$ on the symbols of a pair-string $\boldsymbol{w}$ is a graph with nodes being the symbols of $\boldsymbol{w}$, lying on a line in a certain order, and edges being arcs joining pairs of the same symbol.

Given a pair-string $\boldsymbol{w}$, if in the bipartite graph drawn on the symbols of $\boldsymbol{w}$, the arcs connecting instances of symbol a and instances of symbol b are non-nested (crossed or disjoint) then $\boldsymbol{w}$ is said to be anti-Monge with respect to symbols a, b. If these arcs are nested then it is said to be Monge with respect to symbols a and b. For example, consider pair-strings $\boldsymbol{w} = abab$ and $\boldsymbol{u} = abba$, from the bipartite graph $G_{\boldsymbol{w}}$ shown in Figure 1.4, we can see that $\boldsymbol{w}$ is anti-Monge with respect to symbols a and b. Similarly, from the bipartite graph $G_{\boldsymbol{u}}$, $\boldsymbol{u}$ is Monge with respect to symbols a and b.

Figure 1.4: Figure on the left is $G_{\boldsymbol{w}}$, the bipartite graph for $\boldsymbol{w} = abab$, and Figure on the right is $G_{\boldsymbol{u}}$, the bipartite graph for $\boldsymbol{u} = abba$

Observe that, if a pair-string $\boldsymbol{w}$ is Monge with respect to symbols a and b then $abba$ or $baab$ is a subsequence of $\boldsymbol{w}$, and if $\boldsymbol{w}$ is anti-Monge with respect to a, b then

any of the strings $abab, baba, aabb$, or $bbaa$ is a subsequence of $\boldsymbol{w}$.

For more details on the Monge and anti-Monge condition see [Buss and Soltys, 2013] and [Buss and Yianilos, 1998] and its references. The Monge condition has been widely studied for matching problems and transportation problems. Many problems that satisfy the Monge condition are known to have efficient polynomial time algorithms; for these see [Buss and Yianilos, 1998] and the references cited therein. There are fewer algorithms known for problems that satisfy the anti-Monge property, and some special cases are known to be **NP**-hard [Burkhard et al., 1998]. Motivated by this, in Section 3.3.2 we explore the connection between graphs and strings having Monge and anti-Monge property.

1.5 Background on Complexity

Complexity theory investigates the problem of determining the resources in terms of time, space and more required in order to solve the problem, and "hardness" is a lower bound on resources.

In time complexity theory, problems and classes of problems are studied in terms of the resource 'time'. The running time is measured as the maximum number of steps used by a deterministic Turing Machine on any input of length n. The time analysis of an algorithm is done in the standard way using the O-notation etc. see [Thomas H. Cormen, 2009], [Kleinberg and Tardos, 2006] for a detailed coverage of the topic. The two most referenced classes in time complexity theory are **P** and **NP**.

In space complexity theory, problems and classes of problems are studied in terms of the resource 'space'. The two major classes in space complexity are **PSPACE** and **NSPACE**, and are analogs to **P** and **NP** classes.

We give here a brief introduction of time, space and circuit complexity for completeness. See [Sipser, 2013], [Papadimitriou, 1994], [Soltys, 2009] for a comprehensive understanding of the subject.

Basic Definitions

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. Intuitively, all algorithms with polynomial running time belong to this class. Formally, we define this class as follows: Let $f : \mathbb{N} \to \mathbb{R}$ be a function. Then, TIME($f(n)$), is a time complexity class consisting of all the languages that are decidable by Turing machines running in time $O(f(n))$. Therefore, the class **P** in terms of the class TIME is:

$$\mathbf{P} = \bigcup_k \text{TIME}(n^k)$$

where n is the size of the input and $k \in \mathbb{N}$.

NP is the class of languages that are decidable in polynomial time on a non-deterministic Turing machine. Intuitively, it is the class of problems whose solutions can be verified in polynomial time. NTIME($f(n)$), is a time complexity class consisting of all the languages that are decidable by non-deterministic Turing machines running in time $O(f(n))$. Therefore, the class **NP** in terms of the class NTIME is:

$$\mathbf{NP} = \bigcup_k \text{NTIME}(n^k)$$

NP is the class of languages where the membership in the language can be verified in polynomial time and **P** is the class of languages where the membership in the

language can be tested in polynomial time. We know that $\mathbf{P} \subset \mathbf{NP}$, but whether $\mathbf{P} = \mathbf{NP}$ is still an open problem.

In 1970, Steven Cook in his seminal paper [Cook, 1971], and Leonid Levin in his 1973 paper [Levin, 1973], showed that the complexity of certain problems in the class **NP** is related to the entire class. If polynomial time solutions exists for these problems, then all problems in **NP** would have polynomial time solutions. These problems are called **NP**-complete.

A language A, is said to be **NP**-complete if it satisfies the following conditions:

1. A is in **NP**, and

2. Every B in **NP** is polynomial time reducible to A.

Examples of **NP**-complete problems are: 3-SAT, 3-color, traveling salesman, Exact Cover with 3-Sets and 3-Partition problem.

PSPACE$(f(n))$ is the class of languages decided by an $O(f(n))$ space deterministic Turing machine.

NSPACE$(f(n))$ is the class of languages decided by an $O(f(n))$ space non-deterministic Turing machine.

Savitch's theorem shows that any non-deterministic Turing machine that uses $f(n)$ space can be converted to a deterministic Turing machine using only $f^2(n)$ space. Therefore **PSPACE** = **NSPACE**.

In circuit complexity theory, a *Boolean circuit* is a directed, acyclic, connected graph in which the input nodes are labeled with variables x_i and constants $1, 0$, representing true and false, respectively, and the internal nodes are labeled with standard Boolean connectives $\wedge, \vee, \neg$, that is, AND, OR, NOT, respectively. We often use $\bar{x}$ to denote $\neg x$, and the circuit nodes are often called *gates*.

The *fan-in*, i.e., number of incoming edges, of a $\neg$-gate is always one, and the fan-in of $\wedge, \vee$ can be arbitrary. The *fan-out*, i.e., number of outgoing edges, of any node can also be arbitrary. Note that when the fan-out is restricted to be exactly one, circuits become Boolean formulas. Each node in the graph can be associated with a Boolean function in the obvious way. The function associated with the output gate(s) is the function computed by the circuit. Note that a Boolean formula can be seen as a circuit in which every node has fan-out one, and $\wedge, \vee$ have fan-in 2, and $\neg$ has fan-in one. Thus, a Boolean formula is a Boolean circuit whose structure is tree-like.

The *size* of a circuit is its number of gates, and the *depth* of a circuit is the maximum number of gates on any path from an input gate to an output gate.

A *family of circuits* is an infinite sequence $C = \{C_n\} = \{C_0, C_1, C_2, \ldots\}$ of Boolean circuits where C_n has n input variables. We say that a Boolean predicate P has polysize circuits if there exists a polynomial p and a family C such that $|C_n| \leq p(n)$, and $\forall x \in \{0,1\}^*$, $P(x)$ holds iff $C_{|x|}(x) = 1$.

Let $\mathbf{P}$/poly be the class of all those predicates which have polysize circuit families. It is a standard result in complexity that all predicates in $\mathbf{P}$ have polysize circuits; that is, if a predicate has a polytime Turing machine, it has polysize circuits. The converse of the above does not hold, unless we put a severe restriction on how the n-th circuit is generated; as it stands, there are undecidable predicates that have polysize circuits. The restriction that we place here is that there is a Turing machine that on input 1^n computes $\{C_n\}$ in space $O(\log n)$. This restriction makes a family C of circuits *uniform*.

The predicates (or Boolean functions) that can be decided (or computed) with

polysize, constant fan-in, and depth $O(\log^i n)$ circuits, form the class $\mathbf{NC}^i$. The class $\mathbf{AC}^i$ is defined in the same way, except we allow unbounded fan-in. We set $\mathbf{NC} = \bigcup_i \mathbf{NC}^i$, and $\mathbf{AC} = \bigcup_i \mathbf{AC}^i$, and while it is easy to see that the uniform version of $\mathbf{NC}$ is in $\mathbf{P}$, it is an interesting open question whether they are equal.

We have the following standard result: for all i,

$$\mathbf{AC}^i \subseteq \mathbf{NC}^{i+1} \subseteq \mathbf{AC}^{i+1}.$$

Thus, $\mathbf{NC} = \mathbf{AC}$. Finally, $\mathbf{SAC}^i$ is just like $\mathbf{AC}^i$, except we restrict the $\wedge$ fan-in to be at most two.

Chapter 2

Square-free Words over Alphabet Lists

2.1 Introduction

Thue in his paper [Thue, 1906], was the first to show the existence of square-free words of arbitrary length over a ternary alphabet, by giving a non-erasing square-free morphism (see 1.1). Since his work was not known for a long time, this result was rediscovered by many others independently. For example, the following authors each gave a different square-free morphism over a ternary alphabet: [Aršon, 1937], [Morse and Hedlund, 1944], [Currie, 1991], and [Leech, 1957]. The following morphism ($h : \Sigma \mapsto \Sigma^*$) proposed by Leech over the alphabet $\Sigma = \{1, 2, 3\}$ is uniform (each symbol in Σ is mapped to a same length word in Σ^*) and cyclic (as $h(2)$ can be obtained from $h(1)$ and $h(3)$ from $h(2)$, by a cyclic permutation of symbols of the alphabet).

$$h = \begin{cases} 1 \mapsto 1232132312321 \\ 2 \mapsto 2313213123132 \\ 3 \mapsto 3121321231213 \end{cases} \tag{2.1}$$

In all the square-free morphisms proposed to prove the existence of square-free strings over a ternary alphabet the assumption is that the entire alphabet set is available when a symbol is picked while constructing the square-free string. A natural generalization of this problem is that only subsets of the entire possibly infinite alphabet are available while selecting symbols of the square-free string with a restriction that the size of the subsets remain constant. This generalization has been studied by [Grytczuk et al., 2013, Shallit, 2009, Mhaskar and Soltys, 2015b], among others. In this chapter we study this generalization and investigate the question: Do square-free strings exist over any list $L \in \mathcal{L}_3$? Alternatively, is the class of lists $\mathcal{L}_3$ admissible?

2.1.1 Square-free Strings over List L

Using Lovász Local Lemma (see Section 2.1.2) it has been shown that square-free strings exist over lists where the alphabet size is sufficiently large. The first bound of 64, on the size of the alphabets was given by Alon et al. in [N. Alon and Riordan, 2002]. The best possible bound of four, for the size of alphabets, is given by Grytczuk, Przybylo and Zhu in [J. Grytczuk and Zhu, 2011]. They achieved this tight bound by applying an enhanced version of the Lovász Local Lemma due to [Pegden, 2009]. Grytczuk, Kozik, Micek in [Grytczuk et al., 2013] give a simple argument for the same bound. This simple argument along with the proof has been explained in detail in Appendix A. The question whether square-free strings exist over lists with alphabet

size three still remains open and we investigate this question, i.e., is the class of lists $\mathcal{L}_3$ admissible? We obtain partial results, but the general result remains open.

2.1.2 Lovász Local Lemma

A *probabilistic method* is a way of proving that a structure with certain desired properties exists. In this method we define an appropriate probability space of structures and then show that the desired properties hold in this space with positive probability. This method is a powerful and often used tool in solving many problems in Discrete Mathematics without the need of construction.

The popular Lovász Local Lemma, given by Erdős and Lovász, in the paper [Erdős and Lovász, 1975], is often used in probabilistic methods to show the existence of a property in a given structure. The lemma states that: given a finite set of events in a probability space, if a certain subset of these events, where the events in the subset are mostly independent and have a probability less that one, then the entire subset of events can be avoided with positive probability.

A constructive variant of this lemma with some restrictions applied, is given by Beck in his breakthrough paper [Beck, 1991]. The most recent constructive proof of the local lemma is given by Moser and Tardos in [Moser and Tardos, 2009]. For a detailed discussion on the lemma and the probabilistic method in general see [Alon and Spencer, 2008].

2.1.3 Square-free Strings over $L \in \mathcal{L}_4$

A simple probabilistic way to construct a square-free string over L is as follows: pick any $\boldsymbol{w}_1 \in L_1$, and for $i+1$, assuming that $\boldsymbol{w} = \boldsymbol{w}_1\boldsymbol{w}_2 \ldots \boldsymbol{w}_i$ is square-free, pick a

symbol $a \in L_{i+1}$, and if $\boldsymbol{w}a$ is square-free, then let $\boldsymbol{w}_{i+1} = a$. If, on the other hand, $\boldsymbol{w}a$ has a square $\boldsymbol{vv}$, then $\boldsymbol{vv}$ must be a suffix (as $\boldsymbol{w}$ is square-free by assumption). Delete the right copy of $\boldsymbol{v}$ from $\boldsymbol{w}$, and continue with the resulting string. [Grytczuk et al., 2013], uses this algorithm (see Algorithm 1) to prove the main result of the paper (restated in Theorem 34), which is the existence of square-free strings over any list $L \in \mathcal{L}_4$. In the appendix, we restate the main result of [Grytczuk et al., 2013], and give a detailed explanation of the proof shown in their paper.

Algorithm 1 Erase-Repetition Algorithm

1: $i \leftarrow 1$
2: **while** $i \leq n$ **do**
3: $\quad s_i \leftarrow$ random element of L_i
4: $\quad$ **if** $s_1 s_2 \dots s_i$ is square-free **then**
5: $\quad\quad i \leftarrow i + 1$
6: $\quad$ **else** $\quad\quad \triangleright$ there is exactly one repetition, say $s_{i-2h+1} \dots s_{i-h} s_{i-h+1} \dots s_i$
7: $\quad\quad i \leftarrow i - h + 1$

The correctness of Algorithm 1 in [Grytczuk et al., 2013] also relies on Lemma 2 shown below, which was assumed but not shown in [Grytczuk et al., 2013, line 7 of Algorithm 1, on page 2]. An alternate proof of Lemma 2 can be found in our paper [Mhaskar and Soltys, 2015b].

Lemma 2. *If $\boldsymbol{w}$ is square-free, then for any symbol a, either $\boldsymbol{w}' = \boldsymbol{w}a$ is still square-free, or $\boldsymbol{w}'$ has a unique square (consisting of a suffix of $\boldsymbol{w}'$).*

Proof. The proof is by contradiction. We assume that two distinct squares exists in $\boldsymbol{w}'$ and examine different possibilities of overlap between these two squares and prove that each case leads to a contradiction, thus contradicting the assumption that two squares exist in $\boldsymbol{w}'$.

Suppose that $\boldsymbol{w'} = \boldsymbol{w}a$ has a square; we denote this square as $\boldsymbol{u}a\boldsymbol{u}a$, where $\boldsymbol{u}$ is a subword of $\boldsymbol{w}$. Suppose that there is another square $\boldsymbol{v}a\boldsymbol{v}a$, where $\boldsymbol{v}$ is a subword of $\boldsymbol{w}$. Since the two squares are suffixes of $\boldsymbol{w'}$ and are distinct, one must be the suffix of the other. Without loss of generality we assume that $|\boldsymbol{u}a\boldsymbol{u}| > |\boldsymbol{v}a\boldsymbol{v}|$. The different cases of the overlap are examined below:

1. $\boldsymbol{u} = \boldsymbol{p}\boldsymbol{v}a\boldsymbol{v}$ as shown in Figure 2.1: Here $\boldsymbol{p} \neq \boldsymbol{v}$ and $\boldsymbol{p} \neq a$ as otherwise it would create a repetition in $\boldsymbol{w}$. Then, $\boldsymbol{u}a\boldsymbol{u} = \boldsymbol{p}\boldsymbol{v}a\boldsymbol{v}a\boldsymbol{p}\boldsymbol{v}a\boldsymbol{v}$. This indicates that $\boldsymbol{w}$ has a square '$\boldsymbol{v}a\boldsymbol{v}a$' and this is a contradiction. When $\boldsymbol{p} = \varepsilon$, we get the case $\boldsymbol{u} = \boldsymbol{v}a\boldsymbol{v}$

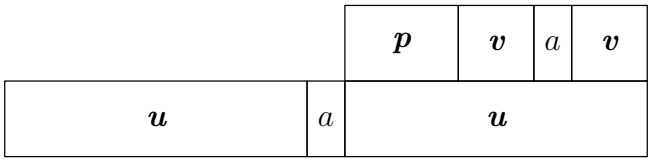

Figure 2.1: $\boldsymbol{u} = \boldsymbol{p}\boldsymbol{v}a\boldsymbol{v}$

   and the above contradiction still holds.

2. $a\boldsymbol{u} = \boldsymbol{v}a\boldsymbol{v}$: Then the first symbol of $\boldsymbol{v}$ is a, i.e., $\boldsymbol{v}_1 = a$. This causes a repetition 'aa' in $\boldsymbol{v}a\boldsymbol{v}$ and therefore a repetition in $\boldsymbol{w}$ and this is a contradiction.

3. $\boldsymbol{u}a\boldsymbol{u} = \boldsymbol{q}\boldsymbol{v}a\boldsymbol{v}$ and $|a\boldsymbol{u}| < |\boldsymbol{v}a\boldsymbol{v}|$ as shown in Figure 2.2: $\boldsymbol{s} \neq \varepsilon$ (otherwise, $\boldsymbol{u} = \boldsymbol{v}$, a contradiction) and $\boldsymbol{p} \neq \varepsilon$ (otherwise, it contradicts the assumption $|a\boldsymbol{u}| < |\boldsymbol{v}a\boldsymbol{v}|$). From Figure 2.2, we have $\boldsymbol{v} = \boldsymbol{p}a\boldsymbol{s}$ and $\boldsymbol{u} = \boldsymbol{s}a\boldsymbol{v}$. Substituting $\boldsymbol{v}$ in $\boldsymbol{u}$ we get, $\boldsymbol{u} = \boldsymbol{s}a\boldsymbol{p}a\boldsymbol{s}$ and $\boldsymbol{u}a\boldsymbol{u} = \boldsymbol{s}a\boldsymbol{p}a\boldsymbol{s}a\boldsymbol{s}a\boldsymbol{p}a\boldsymbol{s}$. Then $\boldsymbol{u}a\boldsymbol{u}$ has a repetition '$a\boldsymbol{s}a\boldsymbol{s}$' and therefore $\boldsymbol{w}$ has a repetition. This is a contradiction.

We can therefore conclude that $\boldsymbol{w'}$ is either square-free or has a unique square. □

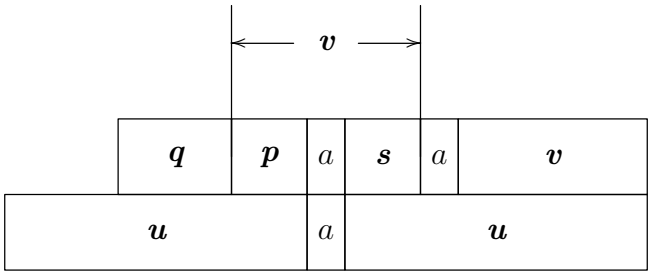

Figure 2.2: $\boldsymbol{uau} = \boldsymbol{qvav}$ and $|a\boldsymbol{u}| < |\boldsymbol{vav}|$

2.2 Alternate Proof of Thue's Result

In this section we give our proof of Thue's result stated in Theorem 1. We restate Theorem 1 below for completeness.

Theorem 1 (restated)**.** *If $\boldsymbol{w} \in \Sigma_3^*$ is a square-free string, then so is $g(\boldsymbol{w})$.*

Proof of Theorem 1. We prove it by induction on $k = |\boldsymbol{w}|$. If $k = 1$, then $\boldsymbol{w} \in \{1, 2, 3\}$, and clearly $g(\boldsymbol{w})$ is square-free in all three cases of g as given in Section 1.4, (1.1). Assume the claim holds for all $\boldsymbol{w}$'s of length k, and consider a square-free $|\boldsymbol{w}| = k+1$, that is, $\boldsymbol{w} = \boldsymbol{w}_1\boldsymbol{w}_2 \ldots \boldsymbol{w}_{k+1}$. Let $\boldsymbol{w'} = g(\boldsymbol{w}) = g(\boldsymbol{w}_1)g(\boldsymbol{w}_2) \ldots g(\boldsymbol{w}_{k+1})$; we want to show that $\boldsymbol{w'}$ is square-free. We argue by contradiction: suppose there exists a $\boldsymbol{v}$ such that $\boldsymbol{vv}$ is a subword of $\boldsymbol{w'}$. Let $\boldsymbol{v}_\ell$ be the left copy of $\boldsymbol{v}$ in $\boldsymbol{w'}$ and let $\boldsymbol{v}_r$ be the right copy of $\boldsymbol{v}$ in $\boldsymbol{w'}$, i.e., $\boldsymbol{vv} = \boldsymbol{v}_\ell\boldsymbol{v}_r$ a subword of $\boldsymbol{w'}$.

Therefore:

$$\boldsymbol{w'} = \boldsymbol{x}\boldsymbol{v}_\ell\boldsymbol{v}_r\boldsymbol{y}.$$

Observe that $g(\boldsymbol{w}_1)$ cannot be a prefix of $\boldsymbol{x}$, and $g(\boldsymbol{w}_{k+1})$ cannot be a suffix of $\boldsymbol{y}$, since in that case we would be able to obtain a repetitive string by applying the morphism to a string $\boldsymbol{w}$ of length $\leq k$, contradicting our inductive assumption.

Thus, $\boldsymbol{x}$ is a non empty proper prefix of $g(\boldsymbol{w}_1)$ and $\boldsymbol{y}$ is a non empty proper suffix of $g(\boldsymbol{w}_{k+1})$. Let the suffix of $g(\boldsymbol{w}_1)$ that coincides with a prefix of $\boldsymbol{v}_\ell$ be denoted by $\boldsymbol{s}$, and let the prefix of $g(\boldsymbol{w}_{k+1})$ that coincides with the suffix of $\boldsymbol{v}_r$ be denoted by $\boldsymbol{p}$. Let $g(\boldsymbol{w}_i)$ be the word containing the first symbol of $\boldsymbol{v}_r$. Finally, let $\boldsymbol{t}$ be the suffix of $\boldsymbol{v}_\ell$ that coincides with a prefix of $g(\boldsymbol{w}_i)$ (note that it may be the case that $\boldsymbol{t} = \varepsilon$), and let $\boldsymbol{u}$ be the prefix of $\boldsymbol{v}_r$ that coincides with a suffix of $g(\boldsymbol{w}_i)$ (by definition, $\boldsymbol{u} \neq \varepsilon$). For a summary of all these relationships see Figure 2.3.

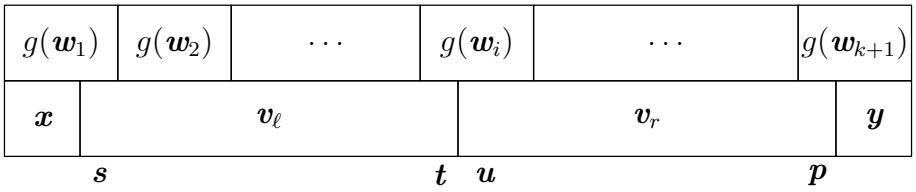

Figure 2.3: The strings $\boldsymbol{w}' = g(\boldsymbol{w}_1)g(\boldsymbol{w}_2)\ldots g(\boldsymbol{w}_{k+1}) = \boldsymbol{x}\boldsymbol{v}_\ell\boldsymbol{v}_r\boldsymbol{y}$.

We consider the possible lengths of $\boldsymbol{s}$, and the possible corresponding values of $g(\boldsymbol{w}_1), g(\boldsymbol{w}_i), g(\boldsymbol{w}_{k+1})$, as shown in Figure 2.3, and taken from (1.1). We derive a contradiction in each case, and conclude that the situation presented in Figure 2.3 is not possible, and hence $g(\boldsymbol{w})$ is square-free.

1. Suppose that $|\boldsymbol{s}| = 1$. Then, $\boldsymbol{s} = 2$, as that is the only suffix of (1.1) of length 1. We now want to show that the first symbol of $\boldsymbol{u}$ equals the first symbol of $\boldsymbol{y}$ (that is, $\boldsymbol{u}_1 = \boldsymbol{y}_1$). Once we have that, we can shift $\boldsymbol{v}_\ell\boldsymbol{v}_r$ one position to the right, and still have a square, albeit in $g(\boldsymbol{w}_2 \ldots \boldsymbol{w}_{k+1})$, contradicting the inductive assumption. First note that $\boldsymbol{u}_1 = \boldsymbol{s} = 2$, so all we have to show is that $\boldsymbol{y}_1 = 2$. But if $\boldsymbol{u}_1 = 2$, it follows that $\boldsymbol{t}$ must be one of the following:

   (a) $g(\boldsymbol{w}_i) = 12312$ and so $\boldsymbol{t} = 1$ or $\boldsymbol{t} = 1231$. If $\boldsymbol{t} = 1$, then it follows that the

prefix of $g(\boldsymbol{w}_2)$ is 312, which is not possible, as no string in (1.1) starts with 3. Hence it must be the case that $\boldsymbol{t} = 1231$, but then 1231 is the block of symbols immediately preceding $\boldsymbol{y}$, which forces the first symbol of $\boldsymbol{y}$ to be 2.

(b) $g(\boldsymbol{w}_i) = 131232$ and so $\boldsymbol{t} = 131$ or $\boldsymbol{t} = 13123$. Since $\boldsymbol{t}$ is a block immediately preceding $\boldsymbol{y}$, it follows that the first symbol of $\boldsymbol{y}$ must be 2.

(c) $g(\boldsymbol{w}_i) = 1323132$ and so $\boldsymbol{t} = 13$ or $\boldsymbol{t} = 132313$. This is similar to subcase (1a) above. If $\boldsymbol{t} = 13$, then the prefix of $g(\boldsymbol{w}_2)$ is 3132, which is not possible. If $\boldsymbol{t} = 132313$, then again the first symbol of $\boldsymbol{y}$ must be 2.

2. Suppose that $|\boldsymbol{s}| = 2$. Then, $\boldsymbol{s} = 12$ or $\boldsymbol{s} = 32$.

(a) If $\boldsymbol{s} = 12$, then $g(\boldsymbol{w}_1) = 12312$, and so the initial segment of $\boldsymbol{v}_r$ must be 12, and so $g(\boldsymbol{w}_i)$ must be 12312 as well. Further, $\boldsymbol{u}$ can be 12312 (in which case $\boldsymbol{t} = \varepsilon$), or $\boldsymbol{u} = 12$ (in which case $\boldsymbol{t} = 123$). In the former case, 312 must be a prefix of $g(\boldsymbol{w}_2)$, which is not possible. In the latter case, the segment immediately preceding $\boldsymbol{y}$ must be 123, which implies that $\boldsymbol{y}_1\boldsymbol{y}_2 = 12$, and again a shift of $\boldsymbol{v}_\ell\boldsymbol{v}_r$ right by two positions creates a square in $g(\boldsymbol{w}_2 \dots \boldsymbol{w}_{k+1})$, contradicting the inductive assumption.

3. If $|\boldsymbol{s}| \geq 3$, then $\boldsymbol{s}$ identifies $g(\boldsymbol{w}_1)$ uniquely, which in turn identifies $\boldsymbol{t}$ uniquely, which in turn identifies the first $|\boldsymbol{s}|$ many symbols of $\boldsymbol{y}$ uniquely, and shows that

$$\boldsymbol{u}_1 \dots \boldsymbol{u}_{|s|} = \boldsymbol{y}_1 \dots \boldsymbol{y}_{|s|}.$$

This means that shifting $\boldsymbol{v}_\ell\boldsymbol{v}_r$ $|\boldsymbol{s}|$ many symbols to the right creates a square in

$g(\boldsymbol{w}_2 \dots \boldsymbol{w}_{k+1})$, contradicting the inductive hypothesis.

This ends the proof. □

Our proof of Thue's result differs from [Smyth, 2003, Section 3.3, pg. 72] in the fact that we use induction on the length of $\boldsymbol{w}$ as opposed to introducing and using blocks, initiators and terminators.

In [Berstel, 1979], Berstel gave the first characterization of square-free morphisms over three letter alphabet. This characterization requires the given morphism to be square-free for all square-free words of size three. This result can also be found in [Berstel, 1995]. In [Crochemore, 1982], Crochemore improved the result in the general case (not only for words of size three). His characterization requires only the resulting words of length five, after application of the morphism to square-free words, to be square-free. He also shows that the bound five given in the characterization is optimal and that such a result does not exist on an alphabet of size four or more.

2.3 Admissible Classes of Lists

In Section 2.1.1, we point to various results showing admissibility of lists. The best bound given so far is by Grytczuk et al. in [Grytczuk et al., 2013]. They show that the class of lists $\mathcal{L}_4$ is admissible and conjecture that the class $\mathcal{L}_3$ is admissible. In [Mhaskar and Soltys, 2015b, Theorem 8], we show that certain (large) subclasses of $\mathcal{L}_3$ are admissible (Theorem 9), and in Chapter 5 [Mhaskar and Soltys, 2015b, Section 4] we propose different approaches for proving this conjecture in its full generality.

Corollary 3 (A. Thue, [Thue, 1906])**.** $\mathcal{L}_{\Sigma_3}$ *is admissible.*

Proof. Consider any $L \in \mathcal{L}_{\Sigma_3}$, where $L = L_1, L_2, \ldots L_n$, and for all $i \in [n]$, $L_i = \Sigma_3$.

Let $\mathcal{T}_1 = 1$, and for all $i > 1$, let $\mathcal{T}_{i+1} = g(\mathcal{T}_i)$. We show by induction on i that $\mathcal{T}_i$ is a sequence of square-free strings of growing length. The basis case is trivial, and the induction step follows from Theorem 1. Also, since g is a non-erasing morphism, $|\mathcal{T}_i| < |\mathcal{T}_{i+1}|$, for all $i \in [n]$. Note that the prefix of any square-free string is also square-free (in fact the same applies to any subword — but not subsequence).

In order to generate a square-free string of length n, we generate $\mathcal{T}_i$ such that $|\mathcal{T}_i| \geq n$, and we then take its prefix of length n. This is our square-free $\boldsymbol{w}$ over L. □

A System of Distinct Representatives (SDR) of a collection of sets $\{L_1, L_2, \ldots, L_n\}$ is a selection of n distinct elements $\{a_1, a_2, \ldots, a_n\}$, $a_i \in L_i$.

Claim 4. *If L has an SDR, then L is admissible.*

Proof. Simply let $\boldsymbol{w} = \boldsymbol{w}_1 \boldsymbol{w}_2 \ldots \boldsymbol{w}_n$, where $\boldsymbol{w}_i = a_i$, for all $i \in [n]$, be the string consisting of the distinct representatives; as all symbols are distinct, $\boldsymbol{w}$ is necessarily square-free. □

It is a celebrated result of P. Hall ([Hall, 1987]) that a necessary and sufficient condition for a collection of sets to have an SDR is that they have the *union property*.

A set $L = \{L_1, L_2, \ldots, L_n\}$ is said to have the union property if for any sub-collection $S = \{S_1, S_2, \ldots, S_k\}$, where $1 \leq k \leq n$, and $S_i \in L$ for all $i \in [k]$, the following property holds: $|S_1 \cup S_2 \cup \ldots \cup S_k| \geq k$.

Corollary 5. *If L has the union property, then L is admissible.*

Given a list L, we say that the mapping $\Phi : L \longrightarrow \Sigma_3$, $\Phi = \langle \phi_i \rangle$, is *consistent* if for all i, $\phi_i : L_i \longrightarrow \Sigma_3$ is a bijection, and for all $i \neq j$, if $a \in L_i \cap L_j$, then $\phi_i(a) = \phi_j(a)$.

In other words, Φ maps all the alphabets to the single alphabet Σ_3, in such a way that the same symbol is always mapped to the same unique symbol in $\Sigma_3 = \{1, 2, 3\}$.

For example, if $L = \{\{a, b, c\}, \{a, c, d\}, \{a, b, e\}, \{a, d, e\}\}$, then the mapping $a \to 1, b \to 2, c \to 3, d \to 2, e \to 3$ is a *consistent mapping.*

Lemma 6. *If L has a consistent mapping, then L is admissible.*

Proof. Suppose that L has a consistent mapping $\Phi = \langle \phi_i \rangle$. By Corollary 3 we pick a square-free $\boldsymbol{w} = \boldsymbol{w}_1 \boldsymbol{w}_2 \dots \boldsymbol{w}_n$ of length n over Σ_3. Let

$$\boldsymbol{w}' = \phi_1^{-1}(\boldsymbol{w}_1)\phi_2^{-1}(\boldsymbol{w}_2) \dots \phi_n^{-1}(\boldsymbol{w}_n),$$

then $\boldsymbol{w}'$ is a string over L, and it is also square-free. If it were the case that $\boldsymbol{vv}$ was a subword of $\boldsymbol{w}'$, then $\boldsymbol{vv}$ under Φ would be a square in $\boldsymbol{w}$, which is a contradiction. □

Let CMP $= \{\langle L \rangle : L$ has a consistent mapping$\}$ be the "Consistent Mapping Problem," i.e., the language of lists $L = L_1, L_2, \dots, L_n$ which have a consistent mapping. We show in Lemma 7 that this problem is **NP**-complete. It is clearly in **NP** as a given mapping can be verified efficiently for consistency.

Lemma 7. CMP *is* **NP***-hard.*

Proof. A graph $G = (V, E)$ is 3-colorable if there exists an assignment of three colors to its vertices such that no two vertices with the same color have an edge between them. The problem 3-color is **NP**-hard, and by [Garey et al., 1976] it remains **NP**-hard even if the graph is restricted to be planar.

We show that CMP is **NP**-hard by reducing the 3-colorability of planar graphs to CMP. Given a planar graph $P = (V, E)$, we first find all its triangles. There are

at most $\binom{n}{3} \approx O(n^3)$ such triangles, and note that two different triangles may have 0, 1, or 2 vertices in common. If the search yields no triangles in P, then by [Grötzsch, 1959] such a P is 3-colorable, and so we map P to a fixed list with a consistent mapping, say $L = L_1 = \{a, b, c\}$. (In fact, by [Dvořák et al., 2011] it is known that triangle-free planar graphs can be colored in linear time.) Otherwise, denote each triangle by its vertices, and let $T_1, T_2, \ldots, T_k$ be the list of all the triangles, each $T_i = \{v_1^i, v_2^i, v_3^i\}$; note that triangles may overlap. We say that an edge $e = (v_1, v_2)$ is *inside* a triangle if both v_1, v_2 are in some T_i. For every edge $e = (v_1, v_2)$ *not* inside a triangle, let $E = \{e, v_1, v_2\}$. Let $E_1, E_2, \ldots, E_\ell$ be all such triples, and the resulting list is:

$$L_P = T_1, T_2, \ldots, T_k, E_1, E_2, \ldots, E_\ell.$$

See example given in Figure 2.4.

We show that L_P has a consistent mapping if and only if P is 3-colorable.

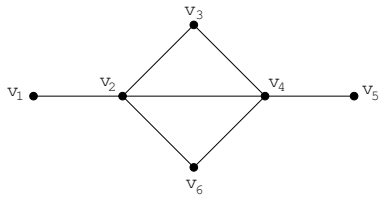

Figure 2.4: In this case the list L_P is composed as follows: there are two triangles, $\{v_2, v_3, v_4\}, \{v_2, v_6, v_4\}$, and there are two edges not inside a triangle giving rise to $\{v_1, v_2, (v_1, v_2)\}, \{v_4, v_5, (v_4, v_5)\}$. Note that this planar graph is 3-colorable: $v_1 \mapsto 1$, $v_2 \mapsto 2$, $v_3 \mapsto 3$, $v_6 \mapsto 3$, $v_4 \mapsto 1$, and $v_5 \mapsto 2$. And the same assignment can also be interpreted as a consistent mapping of the list L_P.

Suppose that P is 3-colorable. Let the colors be labeled with $\Sigma_3 = \{1, 2, 3\}$; each vertex in P can be labeled with one of Σ_3 so that no edge has end-points labeled with the same color. This clearly induces a consistent mapping as each triangle

$T_i = \{v_1^i, v_2^i, v_3^i\}$ gets 3 colors, and each $E = \{e, v_1, v_2\}$ gets two colors for v_1, v_2, and we give e the third remaining color.

Suppose, on the other hand, that L_P has a consistent mapping. This induces a 3-coloring in the obvious way: each vertex inside a triangle gets mapped to one of the three colors in Σ_3, and each vertex not in a triangle is either a singleton, in which case it can be colored arbitrarily, or the end-point of an edge not inside a triangle, in which case it gets labeled consistently with one of Σ_3. As the reduction from P to L_P can be accomplished in polynomial time, it follows that CMP is **NP**-hard. □

We say that a collection of sets $\{L_1, L_2, \ldots, L_n\}$ is a *partition* if for all i, j, $L_i = L_j$ or $L_i \cap L_j = \emptyset$.

Corollary 8. *If L is a partition, then L is admissible.*

Proof. We show that when L is a partition, we can construct a consistent Φ, and so, by Lemma 6, L is admissible. For each i in $[n]$ in increasing order, if L_i is new i.e., there is no $j < i$, such that $L_i \neq L_j$, then let $\phi_i : L_i \longrightarrow \Sigma_3$ be any bijection. If, on the other hand, L_i is not new, there is a $j < i$, such that $L_i = L_j$, then let $\phi_i = \phi_j$. Clearly $\Phi = \langle \phi_i \rangle$ is a consistent mapping. □

Note that by Lemma 6, the existence of a consistent mapping guarantees the existence of a square-free string. The inverse relation does not hold: a list L may not have a consistent mapping, and still be admissible. For example, consider $L = \{\{a, b, c\}, \{a, b, e\}, \{c, e, f\}\}$. Then, in order to have consistency, we must have $\phi_1(a) = \phi_2(a)$ and $\phi_1(b) = \phi_2(b)$. In turn, by bijectivity, this implies that $\phi_1(c) = \phi_2(e)$. Again, by consistency:

$$\phi_3(c) = \phi_1(c) = \phi_2(e) = \phi_3(e),$$

and so $\phi_3(c) = \phi_3(e)$, which violates bijectivity. Hence L does not have a consistent mapping, but $w = abc \in L^+$, and w is square-free.

Let $\mathcal{L}_{\text{SDR}}, \mathcal{L}_{\text{Union}}, \mathcal{L}_{\text{Consist}}$, and $\mathcal{L}_{\text{Part}}$ be classes consisting of lists with: an SDR, the union property, a consistent mapping, the partition property, respectively. Summarizing the results in the above lemmas we obtain the following theorem.

Theorem 9. *$\mathcal{L}_{\text{SDR}}, \mathcal{L}_{\text{Union}}, \mathcal{L}_{\text{Consist}}$, and $\mathcal{L}_{\text{Part}}$ are all admissible.*

For ease of reference, in Section 2.4, we give a table summarizing the notation for classes with different properties and their admissibility.

2.4 Applications

The applications of this generalization of square-free strings can be seen in non-repetitive coloring of graphs and online games. A non-repetitive coloring of a path, is a coloring of its vertices such that the sequence of colors along the path does not contain two identical consecutive blocks. The ideas behind the erase-repetition algorithm [Grytczuk et al., 2013] also led to the result in [Kozik and Micek, 2013] that for every tree and lists of size four one can choose a coloring with no three consecutive identical blocks on any simple path.

In the area of non-repetitive games, using the same arguments of the proof of [Grytczuk et al., 2013, Theorem 1], Grytczuk et al. show that one can generate arbitrarily long square-free sequences, if every second symbol is picked by an adversary, provided the number of available symbols is at least twelve. A tighter bound on an alphabet of size eight is shown in [J. Grytczuk and Micek, 2011].

We propose an online game version of the problem, where the list L is presented

piecemeal, one alphabet at a time, and we select the next symbol without knowing the future, and once selected, we cannot change it later. More precisely, the L_i's are presented one at a time, starting with L_1, and when L_i is presented, we must select $\boldsymbol{w}_i \in L_i$, without knowing $L_{i+1}, L_{i+2}, \ldots$, and without being able to change the selections already committed to in $L_1, L_2, \ldots, L_{i-1}$.

We present the online problem in a game-theoretic context. Given a class of lists $\mathcal{L}$, and a positive integer n, the players take turns, with the adversary starting the game. In the i-th round, the *adversary* presents a set L_i, and the *player* selects a $\boldsymbol{w}_i \in L_i$; the first i rounds of the game can be represented as:

$$G = L_1, \boldsymbol{w}_1, L_2, \boldsymbol{w}_2, \ldots, L_i, \boldsymbol{w}_i.$$

The condition imposed on the adversary is that $L = L_1, L_2, \ldots, L_n$ must be a member of $\mathcal{L}$.

The player has a *winning strategy* for $\mathcal{L}$, if $\forall L_1 \exists \boldsymbol{w}_1 \forall L_2 \exists \boldsymbol{w}_2 \ldots \forall L_n \exists \boldsymbol{w}_n$, such that $L = L_1, L_2, \ldots, L_n \in \mathcal{L}$ and $\boldsymbol{w} = \boldsymbol{w}_1 \boldsymbol{w}_2 \ldots \boldsymbol{w}_n$ is square-free. For example, the player does not have a winning strategy for any L in $\mathcal{L}_{\Sigma_1}$ and $\mathcal{L}_{\Sigma_2}$; see Figure 2.5. On the other hand, the player has a winning strategy for $\mathcal{L}_{\Sigma_3}$: simply pre-compute a square-free $\boldsymbol{w}$, and select $\boldsymbol{w}_i$ from L_i. However, this is not a bona fide online problem (see [Soltys, 2012]), as all future L_i's are known beforehand. In a true online problem we expect the adversary to have the ability to "adjust" the selection of the L_i's based on the history of the game.

We present another class of lists for which the player has a winning strategy. Let $\text{size}_L(i) = |L_1 \cup \ldots \cup L_i|$. We say that L has the *growth* property if for all $1 \leq i < n = |L|$, $\text{size}_L(i) < \text{size}_L(i+1)$. We denote the class of lists with the growth

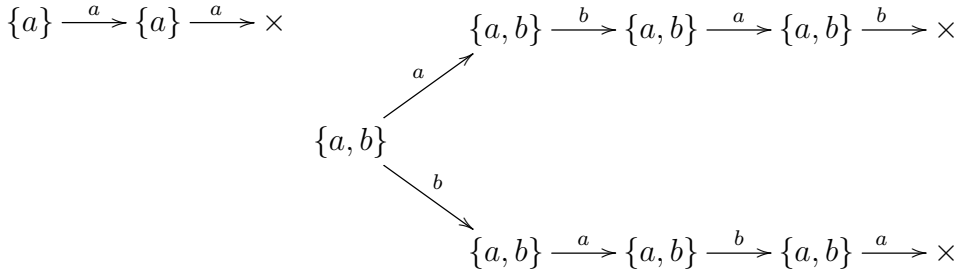

Figure 2.5: Player loses if adversary is allowed subsets of size less than 3: the moves of the adversary are represented with subsets $\{a\}$ and $\{a, b\}$ and the moves of the player are represented with labeled arrows, where the label represents the selection from a subset.

property as $\mathcal{L}_{\text{Grow}}$.

Lemma 10. *The player has a winning strategy for* $\mathcal{L}_{\text{Grow}}$.

Proof. In the i-th iteration, select $\boldsymbol{w}_i$ that has not been selected previously; the existence of such a $\boldsymbol{w}_i$ is guaranteed by the growth property. □

The growth property places a rather strong restriction on L, as it allows the construction of square-free strings where all the symbols are different, and hence they are trivially square-free. Note that the growth property implies the union property discussed in Corollary 5. To see this, note that the growth property implies the existence of an SDR (discussed in Claim 4).

Summary of Classes of Lists

$\mathcal{L}$ denotes a class of lists
$L = L_1, L_2, \ldots, L_n$ denotes a (finite) list of alphabets
L_i denotes a finite alphabet

Class name	**Description**	**Admissible**
$\mathcal{L}_{\Sigma_k}$	for all $i \in [n]$, $L_i = \Sigma_k$	for Σ_k, yes for $k \geq 3$; no for $k < 3$
$\mathcal{L}_k$	for all $i \in [n]$, $\lvert L_i \rvert = k$	yes for $k \geq 4$; no for $k \leq 2$; for $k = 3$?
$\mathcal{L}_{\text{SDR}}$	L has an SDR	yes
$\mathcal{L}_{\text{Union}}$	L has the union property	yes
$\mathcal{L}_{\text{Consist}}$	L has a consistent mapping	yes
$\mathcal{L}_{\text{Part}}$	L is a partition	yes
$\mathcal{L}_{\text{Grow}}$	for all i, $\lvert \cup_{j=1}^{i} L_j \rvert < \lvert \cup_{j=1}^{i+1} L_j \rvert$	yes, even for online games

Chapter 3

String Shuffle

3.1 Introduction

In this area of String Algorithms, we investigate properties of string shuffle (see Section 1.4). Particularly, we study the relationship between graphs and strings exhibiting shuffle properties. The result of this research can be found in our paper [Mhaskar and Soltys, 2015c], which is an expansion of the previously published paper [Soltys, 2013].

Interest in the shuffle operation gained momentum in the last thirty years and the initial work on shuffles arose out of abstract formal languages. Shuffles were later motivated by applications to modeling sequential execution of concurrent processes. The shuffle operation was first used in formal languages by Ginsburg and Spanier [Ginsburg and Spanier, 1965]. Early research with applications to concurrent processes can be found in [Riddle, 1973, Riddle, 1979] and Shaw [Shaw, 1978]. A number of authors, including [Gischer, 1981], [Gruber and Holzer, 2009], [Jantzen, 1981], [Jantzen, 1985], [Jedrzejowicz, 1999], [Jedrzejowicz and Szepietowski, 2001], [Jedrzejowicz and

Szepietowski, 2005], [Mayer and Stockmeyer, 1994], [Shoudai, 1992] and [Ogden et al., 1978] have subsequently studied various aspects of the complexity of the shuffle and iterated shuffle operations in conjunction with regular expression operations and other constructions from the theory of programming languages.

In the early 1980's, Mansfield [Mansfield, 1982, Mansfield, 1983], and Warmuth and Haussler [Warmuth and Haussler, 1984], studied the computational complexity of the shuffle operator on its own. The paper [Mansfield, 1982] gave a dynamic programming algorithm for deciding the following shuffle problem: given input strings $\boldsymbol{u}$, $\boldsymbol{v}$ and $\boldsymbol{w}$, is $\boldsymbol{w} = \boldsymbol{u} \odot \boldsymbol{v}$? The running time complexity of this algorithm is $O(|\boldsymbol{w}|^2)$.

In [Mansfield, 1983] this was extended to give polynomial time algorithms for deciding whether a string $\boldsymbol{w}$ can be written as the shuffle of k strings $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_k$, for a *constant* integer k. The paper [Mansfield, 1983] further proved that if k is allowed to vary, then the problem becomes **NP**-complete (via a reduction from Exact Cover with 3-Sets).

Warmuth and Haussler [Warmuth and Haussler, 1984] gave an independent proof of the above result, and went on to give a rather striking improvement by showing that this problem remains **NP**-complete even if the k strings $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_k$ are equal. That is to say, the question of, given strings $\boldsymbol{u}$ and $\boldsymbol{w}$, whether $\boldsymbol{w}$ is equal to an *iterated shuffle* of $\boldsymbol{u}$ is **NP**-complete. Their proof used a reduction from 3-Partition.

In [Buss and Soltys, 2013] it is shown that square shuffle, i.e., the problem of determining whether a given string $\boldsymbol{w}$ is a shuffle of some $\boldsymbol{u}$ with itself, that is, whether $\boldsymbol{w} = \boldsymbol{u} \odot \boldsymbol{u}$, is **NP**-hard. [Rizzi and Vialette, 2013] gives an alternate proof of the same result.

Mansfield Algorithm

In [Mansfield, 1982], Mansfield gave a polynomial time dynamic programming algorithm for shuffle, that is, given input strings $\boldsymbol{u}, \boldsymbol{v}$ and $\boldsymbol{w}$, the algorithm answers the question: is $\boldsymbol{w} = \boldsymbol{u} \odot \boldsymbol{v}$, in polynomial time. For dynamic programming technique see, [Thomas H. Cormen, 2009], [Kleinberg and Tardos, 2006] or [Soltys, 2012].

The basic idea of the algorithm is to construct a grid graph, with $(|\boldsymbol{u}|+1) \times (|\boldsymbol{v}|+1)$ nodes; the lower-left node is represented with $(0,0)$ and the upper-right node is represented with $(|\boldsymbol{u}|, |\boldsymbol{v}|)$. For any $i < |\boldsymbol{u}|$ and $j < |\boldsymbol{v}|$, we have the edges:

$$\begin{cases} ((i,j),(i+1,j)) & \text{if } \boldsymbol{u}_{i+1} = \boldsymbol{w}_{i+j+1} \\ ((i,j),(i,j+1)) & \text{if } \boldsymbol{v}_{j+1} = \boldsymbol{w}_{i+j+1}. \end{cases} \tag{3.1}$$

Note that both edges may be present, which is what introduces the element of nondeterminism in the traversal of the graph, and reflects the fact that two strings can be shuffled in many ways that initially may seem promising to form $\boldsymbol{w}$. This graph can be interpreted as follows: a path starts at $(0,0)$, and the i-th time it goes up we pick $\boldsymbol{u}_i$, and the j-th time it goes right we pick $\boldsymbol{v}_j$.

Therefore, the algorithm reduces the shuffle problem on strings $\boldsymbol{u}, \boldsymbol{v}$ and $\boldsymbol{w}$ to directed graph reachability. The correctness of the reduction follows from the assertion that given the edges of the grid, defined in (3.1), there is a path from $(0,0)$ to (i,j) if and only if the first $i+j$ bits of $\boldsymbol{w}$ can be obtained by shuffling the first i bits of $\boldsymbol{u}$ and the first j bits of $\boldsymbol{v}$. Thus, node $(|\boldsymbol{u}|, |\boldsymbol{v}|)$ can be reached from node $(0,0)$ if and only if $\boldsymbol{w} = \boldsymbol{u} \odot \boldsymbol{v}$. Given $|\boldsymbol{u}| = |\boldsymbol{v}| = \frac{|w|}{2} = n$, the running time of this algorithm is $O(n^2)$.

For example, consider Figure 3.1. On the left we have a shuffle of 000 and 111 that yields 010101, and on the right we have a shuffle of 011 and 011 that yields 001111. The left instance has a unique shuffle that yields 010101, which corresponds to the unique path from $(0,0)$ to $(3,3)$. On the right, there are several possible shuffles of $011, 011$ that yield 001111 — in fact, eight of them, each corresponding to a distinct path from $(0,0)$ to $(3,3)$.

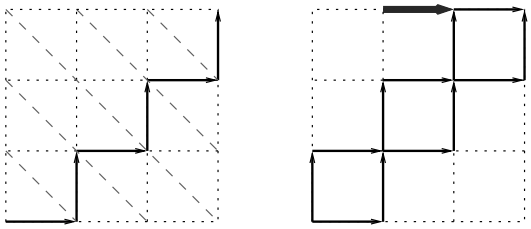

Figure 3.1: On the left we have a shuffle of 000 and 111 that yields 010101, and on the right we have a shuffle of 011 and 011 that yields 001111. The dynamic programming algorithm in [Mansfield, 1982] computes partial solutions along the grey diagonal lines. The thick black arrow in the right diagram is there to symbolize that there are other edges beside the thin black ones; the thick black edge is $((1,3),(2,3))$ and it is there because $\boldsymbol{x}_{1+1} = \boldsymbol{x}_2 = 1 = \boldsymbol{w}_5 = \boldsymbol{w}_{1+3+1}$. The edges are placed according to (3.1).

3.2 Circuit Complexity Bounds for Shuffle

In [Mansfield, 1982], a question was posed of determining a lower complexity bound for the problem. [Soltys, 2013] and [Mhaskar and Soltys, 2015c] give a tight upper and lower bound for the shuffle problem in terms of circuit complexity. Specifically, we show that:

1. bounded depth circuits of polynomial size *cannot* solve shuffle, that is, Shuffle $\notin$

$\mathbf{AC}^0$, but that

2. logarithmic depth circuits of polynomial size *can* do so, that is, Shuffle $\in \mathbf{AC}^1$.

Our proof of the first result relies on the seminal complexity result, due to [Furst et al., 1984], that parity is not in $\mathbf{AC}^0$. We prove that Shuffle $\notin \mathbf{AC}^0$, by reducing parity to shuffle. Since by [Furst et al., 1984], Parity $\notin \mathbf{AC}^0$, it follows that Shuffle $\notin \mathbf{AC}^0$. The result that Shuffle $\in \mathbf{AC}^1$ follows directly from the dynamic programing algorithm given in [Mansfield, 1982] by Mansfield (also discussed in Section 3.1). The detailed proofs of these results can be found in the papers [Soltys, 2013] and [Mhaskar and Soltys, 2015c].

3.3 Further properties of shuffle

Although much progress has been done on understanding shuffle, many questions regarding shuffle remain open. For example, does Shuffle Square (given a string $\boldsymbol{w}$, is it a shuffle of some $\boldsymbol{u}$ with itself), remain **NP**-hard for some alphabets with fewer than seven symbols? See [Buss and Soltys, 2013]. We explore further properties of shuffle in order to better understand the operation.

In the following sections, we show that shuffle can express basic string operations, and many large classes of graphs can be expressed with strings exhibiting shuffle properties.

3.3.1 Expressiveness of shuffle

It is interesting that several different string predicates reduce to shuffle in an easy and natural way. String equality can be expressed as shuffle. Testing if two strings

are equal is equivalent to using the shuffle predicate with the first argument being an empty string, that is ε, and the second and third arguments being the two strings. Formally, $\boldsymbol{u} = \boldsymbol{v} \iff \text{Shuffle}(\varepsilon, \boldsymbol{u}, \boldsymbol{v})$. Shuffling ε with $\boldsymbol{u}$ always results in $\boldsymbol{u}$, and finally checking if $\boldsymbol{v}$ can be obtained from $\boldsymbol{u}$, is equivalent to testing whether $\boldsymbol{u} = \boldsymbol{v}$.

Testing whether a string $\boldsymbol{w}$ is a concatenation of two strings $\boldsymbol{u}$ and $\boldsymbol{v}$ also reduces to shuffle. Let p_0, p_1 be "padding" functions on strings defined as follows:

$$p_0(\boldsymbol{u}) = p_0(\boldsymbol{u}_1\boldsymbol{u}_2\ldots\boldsymbol{u}_n) = 00\boldsymbol{u}_1 00\boldsymbol{u}_2 00\ldots 00\boldsymbol{u}_n 00$$
$$p_1(\boldsymbol{u}) = p_1(\boldsymbol{u}_1\boldsymbol{u}_2\ldots\boldsymbol{u}_n) = 11\boldsymbol{u}_1 11\boldsymbol{u}_2 11\ldots 11\boldsymbol{u}_n 11$$

that is, p_b, $b \in \{0,1\}$ pads the string $\boldsymbol{u}$ with a pair of b's between any two symbols of $\boldsymbol{u}$, as well as a pair of b's before and after $\boldsymbol{u}$.

To express testing whether a string $\boldsymbol{w}$ is a concatenation of strings $\boldsymbol{u}$ and $\boldsymbol{v}$ using shuffle operation, we have our first two inputs as $p_0(\boldsymbol{u})$ and $p_1(\boldsymbol{v})$. The third input is just juxtaposition of the prefix of $\boldsymbol{w}$ of length $|\boldsymbol{u}|$ with the function p_0 applied, and the suffix of $\boldsymbol{w}$ of length $|\boldsymbol{v}|$ with the function p_1 applied. We use "juxtaposition" since we do not want to define concatenation in terms of concatenation. This is summarized in Claim 11. Here, we assume that $|\boldsymbol{w}| = |\boldsymbol{u}| + |\boldsymbol{v}|$ to simplify the argument.

Claim 11. $\boldsymbol{w} = \boldsymbol{u} \cdot \boldsymbol{v}$ *iff*

$$p_0(\boldsymbol{w}_1\boldsymbol{w}_2\ldots\boldsymbol{w}_{|\boldsymbol{u}|})p_1(\boldsymbol{w}_{|\boldsymbol{u}|+1}\boldsymbol{w}_{|\boldsymbol{u}|+2}\ldots\boldsymbol{w}_{|\boldsymbol{u}|+|\boldsymbol{v}|}) = p_0(\boldsymbol{u}) \odot p_1(\boldsymbol{v}).$$

Proof. The direction "$\Rightarrow$" is easy to see; for direction "$\Leftarrow$" we use the following

notation, to make the argument cleaner:

$$\begin{aligned}
\boldsymbol{r} &= p_0(\boldsymbol{u}) = 00\boldsymbol{u}_1 00\ldots \\
\boldsymbol{s} &= p_1(\boldsymbol{v}) = 11\boldsymbol{v}_1 11\ldots \\
\boldsymbol{t} &= p_0(\boldsymbol{w}_1\boldsymbol{w}_2\ldots\boldsymbol{w}_{|\boldsymbol{u}|})p_1(\boldsymbol{w}_{|\boldsymbol{u}|+1}\boldsymbol{w}_{|\boldsymbol{u}|+2}\ldots\boldsymbol{w}_{|\boldsymbol{u}|+|\boldsymbol{v}|}) \\
&= 00\boldsymbol{w}_1 00\ldots 00\boldsymbol{w}_{|\boldsymbol{u}|}0011\boldsymbol{w}_{|\boldsymbol{u}|+1}11\ldots 11\boldsymbol{w}_{|\boldsymbol{u}|+|\boldsymbol{v}|}11
\end{aligned}$$

If $\boldsymbol{t} = \boldsymbol{r} \odot \boldsymbol{s}$, then we must take the first two bits of $\boldsymbol{r}$ (00) in order to cover the first two bits of $\boldsymbol{t}$ (00). If $\boldsymbol{u}_1 = \boldsymbol{w}_1 = 1$, then we could ostensibly take the first bit of $\boldsymbol{s}$ (1), but the bit following $\boldsymbol{w}_1$ is 0, and $\boldsymbol{u}_1 = 1$ and the second bit of $\boldsymbol{s}$ is 1; so taking the first bit of $\boldsymbol{s}$ leads to a dead end. Thus, we must use $\boldsymbol{u}_1$ to cover $\boldsymbol{w}_1$.

We formalize this argument by induction on the length of $\boldsymbol{t}$ as follows: The basis case is trivial, as $\boldsymbol{t}_1 = \boldsymbol{r}_1 = 0$. Suppose for all $k < |\boldsymbol{t}|$ the property holds. We need to show that it holds for $k+1$. We examine two cases:

1. If $k < 2(|\boldsymbol{u}| + 1)$, then by induction hypothesis we know that $\boldsymbol{t}_1\boldsymbol{t}_2\ldots\boldsymbol{t}_k = \boldsymbol{r}_1\boldsymbol{r}_2\ldots\boldsymbol{r}_k$. The only possibilities for the symbols $\boldsymbol{t}_{k+1}\boldsymbol{t}_{k+2}\boldsymbol{t}_{k+3}$ in $\boldsymbol{t}$ are: $100, 000, 001$ and 010. If $\boldsymbol{t} = \boldsymbol{r} \odot \boldsymbol{s}$, we must take the next three symbols of $\boldsymbol{r}$ following k, that is symbols $\boldsymbol{r}_{k+1}\boldsymbol{r}_{k+2}\boldsymbol{r}_{k+3}$, to cover symbols $\boldsymbol{t}_{k+1}\boldsymbol{t}_{k+2}\boldsymbol{t}_{k+3}$ in $\boldsymbol{t}$ as none of the strings $100, 000, 001$ and 010 are subwords of $\boldsymbol{s}$.

2. If $k \geq 2(|\boldsymbol{u}| + 1)$, then by induction hypothesis we know that $\boldsymbol{t}_1\boldsymbol{t}_2\ldots\boldsymbol{t}_k = \boldsymbol{r}_1\boldsymbol{r}_2\ldots\boldsymbol{r}_{2(|\boldsymbol{u}|+1)}\boldsymbol{s}_{k-2|\boldsymbol{u}|-1}\ldots\boldsymbol{s}_k$. For $\boldsymbol{t} = \boldsymbol{r} \odot \boldsymbol{s}$, we must take the remaining bits of $\boldsymbol{s}$ to cover $\boldsymbol{t}$, as we used up all the symbols of $\boldsymbol{r}$ to cover the first $2(|\boldsymbol{u}| + 1)$ symbols of $\boldsymbol{t}$.

It follows that $\boldsymbol{t} = \boldsymbol{r} \odot \boldsymbol{s} \Rightarrow \boldsymbol{t} = \boldsymbol{r} \cdot \boldsymbol{s}$, which in turn implies $\boldsymbol{w} = \boldsymbol{u} \cdot \boldsymbol{v}$. □

3.3.2 Expressing graphs with shuffle

In this section we explore the connection between general graphs, and the shuffle problem. We show that many graphs can be expressed with strings exhibiting shuffle properties. We start by giving a string construction for graphs and show that large classes of graphs can be expressed as strings using this construction.

Construction of strings representing graphs

Given a graph $G = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$, we construct a string $\boldsymbol{w}_G$ where the alphabet set for $\boldsymbol{w}_G$ is denoted as $\Sigma_V = \{v_1, v_2, \ldots, v_n\}$, that is, the string $\boldsymbol{w}_G$ is constructed over the vertices's of the graph G, with the following properties:

1. $\boldsymbol{w}_G$ is a pair-string, that is, each vertex in the set $\Sigma_V = \{v_1, v_2, \ldots, v_n\}$ appears exactly twice in the string $\boldsymbol{w}_G$.

2. The edges in E are represented based on the ordering of the first and second instance of the end points of the edge in $\boldsymbol{w}_G$. Specifically, if (v_1, v_2) is an edge in E, then the arcs connecting the first and second instance of v_1 and v_2 are non nested, that is, $\boldsymbol{w}_G$ is anti-Monge with respect to v_1, v_2 (see Section 1.4). Therefore, exactly one of the following four strings is a subsequence of $\boldsymbol{w}_G$: $v_1^1\, v_2^1\, v_1^2\, v_2^2$, $v_2^1\, v_1^1\, v_2^2\, v_1^2$, $v_1^1\, v_1^2\, v_2^1\, v_2^2$, $v_2^1\, v_2^2\, v_1^1\, v_1^2$. Observe in Figure 3.2, the arcs connecting instances of the same symbol are not nested.

3. If (v_1, v_2) is not an edge in E, then the arcs connecting the first and second instance of v_1 and v_2 are nested, that is, $\boldsymbol{w}_G$ is Monge with respect to v_1, v_2 (see Section 1.4). Therefore, exactly one of the following two strings is a subsequence

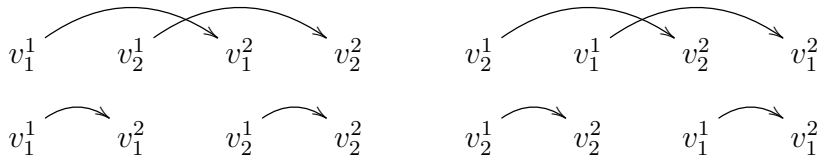

Figure 3.2: Four ways to represent an edge (v_1, v_2) of a graph G in $\boldsymbol{w}_G$.

of $\boldsymbol{w}_G$: $v_1^1\, v_2^1\, v_2^2\, v_1^2$, $v_2^1\, v_1^1\, v_1^2\, v_2^2$. Observe in Figure 3.3, the arcs connecting instances of the same symbol are nested.

Figure 3.3: Two ways to represent *disconnected* nodes v_1 and v_2 of a graph G in $\boldsymbol{w}_G$.

If the string $\boldsymbol{w}_G$ can be constructed with the above properties then we say that $\boldsymbol{w}_G$ *represents* G. We drop the superscripts representing the first and second instance of a vertex in $\boldsymbol{w}_G$ as it clear from the context.

Classes of graphs represented with pair-strings

Many classes of graphs can be represented with the string construction shown in Section 3.3.2. Examples of classes of graphs represented with this construction are: Cliques, Independent set and all graphs with four vertices. Below we give string constructions for Cliques and Independent set.

Let $G = (V, E)$ be a clique, where $V = \{v_1, v_2, \ldots, v_n\} = \Sigma_V$. The string $\boldsymbol{w}_G$ that represents G is:

$$\boldsymbol{w}_G = (v_1 v_2 \ldots v_n)(v_1 v_2 \ldots v_n),$$

An alternate way of representing G as a string w_G is:

$$\boldsymbol{w}_G = v_1 v_1 v_2 v_2 \dots v_n v_n$$

Observe in Figure 3.4, the arcs connecting the first and second instances of every

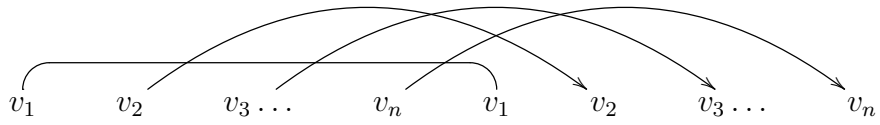

Figure 3.4: String construction $\boldsymbol{w}_G$ for Cliques

symbol in Σ_V is non nested with the arcs connecting the first and second instance of the rest of the symbols in the string $\boldsymbol{w}_G$. Therefore, satisfying the condition that all vertices share an edge between each other in a Clique.

Let $G = (V, E)$ be an independent set, where $V = \{v_1, v_2, \dots, v_n\} = \Sigma_V$. The string $\boldsymbol{w}_G$ that represents G is:

$$\boldsymbol{w}_G = (v_1 v_2 \dots v_n)(v_1 v_2 \dots v_n)^R,$$

where $(\boldsymbol{w}_1 \boldsymbol{w}_2 \dots \boldsymbol{w}_n)^R = \boldsymbol{w}_n \boldsymbol{w}_{n-1} \dots \boldsymbol{w}_1$, i.e., the reverse of a string. A trivial, but alternate way of representing G as a string w_G is:

$$\boldsymbol{w}_G = (v_1 v_2 \dots v_n)^R (v_1 v_2 \dots v_n)$$

Observe in Figure 3.5, the arcs connecting the first and second instances of every

symbol in Σ_V is nested with the arcs connecting the first and second instances of the rest of the symbols in the string $\boldsymbol{w}_G$. Therefore satisfying the condition of an Independent set that no edge exists between any two vertices in the set.

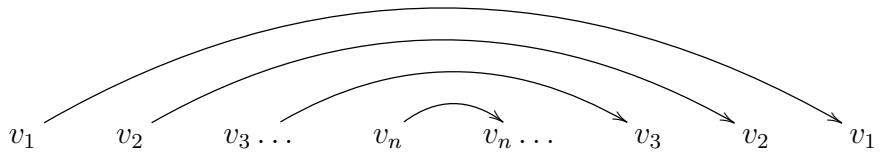

Figure 3.5: String construction $\boldsymbol{w}_G$ of Independent Set

Although we have seen many classes of graphs being represented with strings using the construction given in Section 3.3.2, there are graphs that cannot be represented using this construction. For example, the Hamiltonian cycle G on five vertices shown in Figure 3.6 cannot be represented as a string using this construction.

Suppose $\boldsymbol{w}_G$ is the string representing the Hamiltonian cycle G. Recall that in a pair string representing a graph, an edge (v_1, v_2) in the graph is represented by having any one of the subsequences $v_1v_1v_2v_2$, $v_2v_2v_1v_1$, $v_1v_2v_1v_2$, $v_2v_1v_2v_1$ in $\boldsymbol{w}_G$, and if there is no edge between the vertices v_1, v_2 then either $v_1v_2v_2v_1$ or $v_2v_1v_1v_2$ is a subsequence of $\boldsymbol{w}_G$. In Figure 3.6 we see that (v_1, v_2) (v_2, v_3), (v_3, v_4), (v_4, v_5) and (v_5, v_1) are the edges of the Hamiltonian cycle G. Therefore the following strings cannot be subsequences of $\boldsymbol{w}_G$:

- $v_1v_2v_2v_1$ and $v_2v_1v_1v_2$
- $v_1v_5v_5v_1$ and $v_5v_1v_1v_5$
- $v_2v_3v_3v_2$ and $v_3v_2v_2v_3$

- $v_4v_3v_3v_4$ and $v_3v_4v_4v_3$
- $v_4v_5v_5v_4$ and $v_5v_4v_4v_5$

Also, since the following pairs of vertices (v_1, v_3), (v_1, v_4), (v_2, v_4) , (v_2, v_5), and (v_3, v_5) are not connected by an edge in G, exactly one string from each item in the below list is required to be a subsequence of $\boldsymbol{w}_G$.

- $v_1v_3v_3v_1$ or $v_3v_1v_1v_3$
- $v_1v_4v_4v_1$ or $v_4v_1v_1v_4$
- $v_2v_4v_4v_2$ or $v_4v_2v_2v_4$
- $v_2v_5v_5v_2$ or $v_5v_2v_2v_5$
- $v_3v_5v_5v_3$ or $v_5v_3v_3v_5$

Since it is not possible to satisfy each of the above subsequence requirement, a Hamiltonian cycle G on five vertices cannot be represented with the proposed string representation.

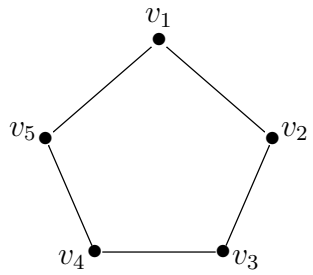

Figure 3.6: The Hamiltonian cycle G on five vertices is the smallest graph without a string representation.

In Lemma 12, we show that a large family of graphs can be represented with strings having shuffle properties.

We denote the concatenation of strings $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n$ by the product notation $\Pi_{i=1}^{n} \boldsymbol{x}_i$, that is, $\Pi_{i=1}^{n} \boldsymbol{x}_i = \boldsymbol{x}_1 \cdot \boldsymbol{x}_2 \cdot \ldots \cdot \boldsymbol{x}_n$. Given a string $\boldsymbol{w}$ of even length, $|\boldsymbol{w}| = n = 2k$, we define $l(\boldsymbol{w})$ to be the left half of $\boldsymbol{w}$ and $r(\boldsymbol{w})$ to be the right half of $\boldsymbol{w}$, i.e., $l(\boldsymbol{w}) = \boldsymbol{w}_1 \boldsymbol{w}_2 \ldots \boldsymbol{w}_k$ and $(\boldsymbol{w}) = \boldsymbol{w}_{k+1} \boldsymbol{w}_{k+2} \ldots \boldsymbol{w}_n$. Thus, $\boldsymbol{w} = l(\boldsymbol{w}) r(\boldsymbol{w})$, i.e., $\boldsymbol{w}$ is the concatenation of its left and right halves.

A rooted tree is a tree with the root singled out. Thus, a rooted tree is represented by a pair (T, R) where T is the tree, and R is the root. Given any node v in T, we let T_v be the subtree of T rooted at v. In particular, the whole tree can be represented as T_R. We view our rooted trees as implicitly directed, where the direction is from the root to the leaves.

Lemma 12. *If G is a graph such that either G or G^c is isomorphic to a transitive closure of a tree, then G can be represented with a string $\boldsymbol{w}_G$.*

Proof. Let T be a rooted tree and consider its representation in Figure 3.7. Here R is the root of T, and $R_1, R_2, \ldots, R_n$ are the children of R, with their respective subtrees $T_{R_1}, T_{R_2}, \ldots, T_{R_n}$. Note that since the tree is considered to be directed, R is (for example) connected to every node in T_{R_1}, but node R_1 is not connected to node R_2.

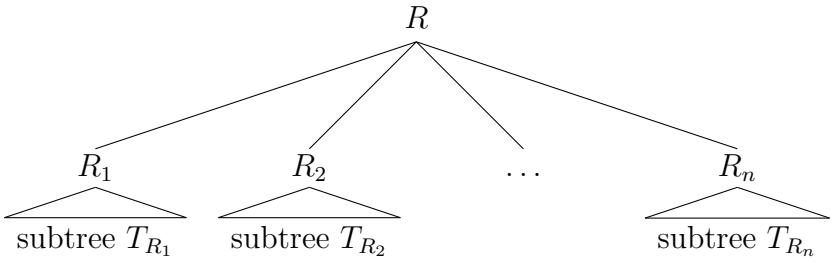

Figure 3.7: Tree T rooted at R.

Let $\hat{T}$ represent the transitive closure of T. Thus, $\hat{T}$ has all the edges of T plus

given any node v in T, v has an edge to every node in T_v (except itself). In particular, following the naming in Figure 3.7, R is connected to every node in T by an edge, and R_i is connected to every node in T_{R_i} by an edge, and so on. Note that if T were not directed, its transitive closure would simply be the clique on its nodes. Since it is directed, a node is connected to any other node on a path from it to some leaf. However, once all edges are computed their direction is dropped, and so $\hat{T}$ is undirected.

We are going to show that given an undirected graph G such that either G or G^c is isomorphic to some $\hat{T}$, the graph G can be represented with a string $\boldsymbol{w}_G$. We show how to construct $\boldsymbol{w}_G$ over Σ_V. Note that we work with three graphs: $G, T, \hat{T}$, where G is undirected, T is implicitly directed, and $\hat{T}$ is obtained from T by a transitive closure, and then the directions on the edges of $\hat{T}$ are dropped.

Suppose that G is isomorphic to some $\hat{T}$, and throughout the construction keep in mind Figure 3.7. We build $\boldsymbol{w}_G = \boldsymbol{w}_{\hat{T}}$ inductively, and our procedure maintains the following property for each subtree: each subtree T_i (i.e., the subtree rooted at R_i) has a corresponding $\boldsymbol{w}_{\hat{T}_i}$ such that $\Sigma_{l(\boldsymbol{w}_{\hat{T}_i})} = \Sigma_{r(\boldsymbol{w}_{\hat{T}_i})}$. Thus, exactly half of the symbols of $\boldsymbol{w}_{\hat{T}_i}$ are in the left half (and therefore, exactly half of the symbols are in the right half of $\boldsymbol{w}_{\hat{T}_i}$). Note that the symbols do *not* necessarily occur in the same order in the two halves.

We are going to construct $\boldsymbol{w}_G = \boldsymbol{w}_{\hat{T}}$ by structural induction on T. In the basis case T consists of a single node R, and $\boldsymbol{w}_{\hat{T}} = RR$. Note that the property $l(w_{\hat{T}}) = l(RR) = R = r(RR) = r(w_{\hat{T}})$ is maintained. In the inductive step we build $w_{\hat{T}}$ as follows:

$$\boldsymbol{w}_G = \boldsymbol{w}_{\hat{T}} = R \left[\Pi_{i=1}^{n} l(\boldsymbol{w}_{\hat{T}_i})\right] R \left[\Pi_{i=n}^{1} r(\boldsymbol{w}_{\hat{T}_i})\right]. \tag{3.2}$$

Note that $\Sigma_{l(\boldsymbol{w}_{\hat{T}})} = \Sigma_{r(\boldsymbol{w}_{\hat{T}})}$, since inductively, for all i, $\Sigma_{l(\boldsymbol{w}_{\hat{T}_i})} = \Sigma_{r(\boldsymbol{w}_{\hat{T}_i})}$, and one copy of R is in $l(\boldsymbol{w}_{\hat{T}})$ and the other in $r(\boldsymbol{w}_{\hat{T}})$. Also note that the product inside the right square brackets of (3.2) is given in reverse order.

Note that the first occurrence of R is to the left of all other symbols, and the second occurrence of R falls in the middle of all the "subtrees"; thus any arc associated with any vertex in the subtree rooted at R overlaps in the right way with the R-arc. On the other hand, the subtrees associated with each $R_1, R_2, \ldots, R_n$ are such that any two arcs from different subtrees are nested. See Figure 3.8.

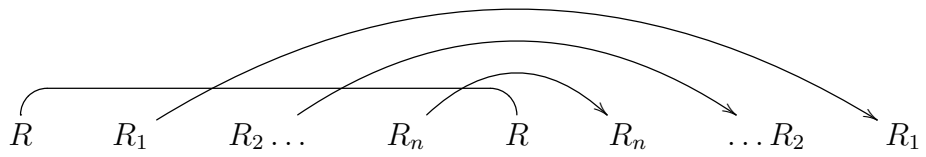

Figure 3.8: Recursive construction of $\boldsymbol{w}_G$. Note that in the left half the R_i's are given in increasing order, and in the right half they are given in decreasing order.

We now give a small example of the construction. Consider the tree T in Figure 3.9. We are going to construct the corresponding $\boldsymbol{w}_{\hat{T}}$. We start with the tree rooted at R_1, where $\boldsymbol{w}_{\hat{T}_1} = R_1R_{11}R_{12}R_1R_{12}R_{11}$, and $\boldsymbol{w}_{\hat{T}_2} = R_2R_{21}R_{22}R_2R_{22}R_{21}$. We now combine, inductively, the two sub-trees rooted at R_1 and R_2 into one tree rooted at R:

$$\begin{aligned}\boldsymbol{w}_{\hat{T}} &= Rl(\boldsymbol{w}_{\hat{T}_1})l(\boldsymbol{w}_{\hat{T}_2})Rr(\boldsymbol{w}_{\hat{T}_2})r(\boldsymbol{w}_{\hat{T}_1}) \\ &= RR_1R_{11}R_{12}R_2R_{21}R_{22}RR_2R_{22}R_{21}R_1R_{12}R_{11}\end{aligned}$$

and now note that, for example, R_2 and R_{12} are not connected in $\hat{T}$, and so arc $(5, 9)$ is

nested under arc $(4, 13)$, where the numbers indicate the position of the corresponding symbol in $\boldsymbol{w}_{\hat{T}}$. On the other hand, R is connected to R_{11} in $\hat{T}$, and so are arcs $(1, 8)$ and $(3, 14)$ are not nested.

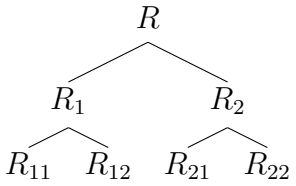

Figure 3.9: Example of a small tree T.

Suppose now that G^c, where an edge is in G^c if and only if it is not in G, is isomorphic to $\hat{T}$. The basis case, where G^c consists of a single node, is the same as in the first case (G is isomorphic to $\hat{T}$). The difference is in the inductive step.

Let $\boldsymbol{w}_{\hat{T}_1^c}, \boldsymbol{w}_{\hat{T}_2^c}, \ldots, \boldsymbol{w}_{\hat{T}_n^c}$ be strings that correspond to the following subtrees: $\hat{T}_1^c, \hat{T}_2^c, \ldots, \hat{T}_n^c$, and hence they represent faithfully the connections in the corresponding vertices in G. We now complete the construction of $\boldsymbol{w}_G$ by adding the root R, so that:

$$\boldsymbol{w}_G = \boldsymbol{w}_{\hat{T}^c} = R(\Pi_{i=1}^n \boldsymbol{w}_{\hat{T}_i^c})R.$$

To see why this works note that if G^c is isomorphic to $\hat{T}$, then G is isomorphic to $\hat{T}^c$, and any arc in $\Pi_{i=1}^n \boldsymbol{w}_{\hat{T}_i^c}$ is nested under the arc between the R's at the ends of the string. This means that R is not connected to any of the nodes in $T_1, T_2, \ldots, T_n$. This ends our proof. □

Chapter 4

A formal framework for Stringology

4.1 Introduction

Many techniques have been developed over the years to prove properties of finite strings, such as suffix arrays, border arrays, and decomposition algorithms such as Lyndon factorization. However, there is no unifying theory or framework, formalizing it. In this chapter, we propose a unifying theory of strings based on a three sorted logical theory, which we call $\mathcal{S}$. By engaging in this line of research, we hope to bring the richness of the advanced field of Proof Complexity to Stringology, and eventually create a unifying theory of strings. Most of the work in this chapter can be found in our paper [Mhaskar and Soltys, 2015a].

The advantage of this approach is that proof theory integrates proofs and computations; this can be beneficial to Stringology as it allows us to extract efficient algorithms from proofs of assertions. More concretely, if we can prove in $\mathcal{S}$ a property

of strings of the form: "for all strings $\boldsymbol{v}$, there exists a string $\boldsymbol{u}$ with property P," i.e., $\exists \boldsymbol{u} \leq I\ P(\boldsymbol{u}, \boldsymbol{v})$, where $|\boldsymbol{u}| \leq I$, then we can mechanically extract an actual algorithm which computes $\boldsymbol{u}$ for any given $\boldsymbol{v}$. For example, suppose that we show that $\mathcal{S}$ proves that every string has a certain decomposition; then, we can actually extract a procedure from the proof for computing such decompositions.

4.2 Background

In *predicate logic* or *single-sorted first order logic*, a *language* (also called *vocabulary*) $\mathcal{L}$ is a set consisting of constant symbols, function symbols, and predicate symbols. If the equality predicate symbol, that is the symbol '=' is in $\mathcal{L}$, then it is always interpreted as the true equality. The $\mathcal{L}$-*Terms* and $\mathcal{L}$-*Formulas* in predicate logic are strings over the symbols of the language and the following additional symbols:

- An infinite set of variables.
- Logical connectives $\neg, \wedge, \vee$; logical constants T, F (for False, True)
- Quantifiers $\forall, \exists$ (for all, there exists)
- (,) (parentheses)

In particular, $\mathcal{L}$-Terms are strings built from variables and function symbols. The basic type of formulas also known as *atomic formulas* are strings built from the predicate symbols applied to $\mathcal{L}$-Terms, and logical constants. $\mathcal{L}$-Formulas are strings built from atomic formulas, logical connectives, quantifiers, and parentheses. A *subformula* β of a formula α is a substring of α that is also a formula. Note that the use of parentheses in formulas is primarily for clarity and unambiguity.

$\mathcal{L}$-Terms and $\mathcal{L}$-Formulas are syntactic objects. The semantics of terms and formulas is given by means of a structure and object assignment. A *structure* $\mathcal{M}$ consists of a non empty set called the *universe of discourse* M (variables in $\mathcal{L}$-Terms and $\mathcal{L}$-Formulas are intended to range over M), and an associated function (relation) for each function (predicate) symbol in $\mathcal{L}$. An *object assignment* τ is a map that maps variables to objects in the universe of discourse. Terms are intended to represent objects in the universe of discourse and formulas evaluate to logical constants true or false. If a formula under the structure holds, that is the object assignment evaluates the formula to true, then the structure is said to be a *model* for the given formula and the formula is said to be *valid*.

A single sorted *theory* over a vocabulary $\mathcal{L}$ is a set of formulas over $\mathcal{L}$ which is closed under logical consequence and universal closure. A *multi sorted theory* such as a *three sorted theory* is an extension of the single sorted first-order logical theory. The language of the three sorted theory consists of constant symbols, function symbols and predicate symbols, for each sort. The infinite set of variables consists of three different types, corresponding to the three different sorts. The terms of a three sorted theory are of three different types corresponding to the three sorts. The formulas of a three sorted theory are defined similar to the single sorted theory, with the only difference being that the given predicate symbol could possibly be applied to terms of any sort.

The structure for the three sorted theory consists of the universe of discourse, and is a tuple consisting of three non empty sets corresponding to each sort and an associated function (relation), with arguments belonging to any of the different sort, for each function (relation) symbol.

We give here a brief introduction to the logical theory for completeness. For a comprehensive understanding of Proof Complexity see [Cook and Nguyen, 2010] which contains a complete treatment of the subject; we follow its methodology and techniques for defining our theory $\mathcal{S}$. We also use some rudimentary λ-calculus from [Soltys and Cook, 2004] to define string constructors in our language.

4.3 Formalizing the theory of finite strings

In this chapter, we propose a three sorted theory that formalizes the reasoning about finite strings. We call our theory $\mathcal{S}$. The three sorts are *indices*, *symbols*, and *strings*. We start by defining a convenient and natural language for making assertions about strings.

4.3.1 The language of strings $\mathcal{L}_{\mathcal{S}}$

Definition 13. *$\mathcal{L}_{\mathcal{S}}$, the language of strings, is defined as follows:*

$$\mathcal{L}_{\mathcal{S}} = [0_{\text{index}}, 1_{\text{index}}, +_{\text{index}}, -_{\text{index}}, \cdot_{\text{index}}, \text{div}_{\text{index}}, \text{rem}_{\text{index}},$$

$$\mathbf{0}_{\text{symbol}}, \sigma_{\text{symbol}}, \text{cond}_{\text{symbol}}, ||_{\text{string}}, e_{\text{string}}; <_{\text{index}}, =_{\text{index}} <_{\text{symbol}}, =_{\text{symbol}}, =_{\text{string}}]$$

The table below explains the intended meaning of each symbol.

Formal	Informal	Intended Meaning
	Index	
0_{index}	0	the integer zero
1_{index}	1	the integer one
$+_{\text{index}}$	$+$	integer addition
$-_{\text{index}}$	$-$	bounded integer subtraction
$\cdot_{\text{index}}$	$\cdot$	integer multiplication (we also just use juxtaposition)
$\text{div}_{\text{index}}$	div	integer division
$\text{rem}_{\text{index}}$	rem	remainder of integer division
$<_{\text{index}}$	$<$	less-than for integers
$=_{\text{index}}$	$=$	equality for integers
	Alphabet symbol	
$\mathbf{0}_{\text{symbol}}$	$\mathbf{0}$	default symbol in every alphabet
σ_{symbol}	σ	unary function for generating more symbols
$<_{\text{symbol}}$	$<$	ordering of alphabet symbols
$\text{cond}_{\text{symbol}}$	cond	a conditional function
$=_{\text{symbol}}$	$=$	equality for alphabet symbols
	String	
$\|\|_{\text{string}}$	$\|\|$	unary function for string length
e_{string}	e	binary fn. for extracting the i-th symbol from a string
$=_{\text{string}}$	$=$	string equality

Note that in practice we use the informal language symbols as otherwise it would be tedious to write terms, but the meaning will be clear from the context. When we write $i \leq j$ we abbreviate the formula $i < j \vee i = j$.

4.3.2 Syntax of $\mathcal{L}_{\mathcal{S}}$

We use metavariables $i, j, k, l, \ldots$ to denote indices, metavariables $a, b, c, \ldots$ to denote alphabet symbols, and metavariables $\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}, \ldots$ to denote strings. When a variable can be of any type, i.e., a meta-meta variable, we write it as $X, Y, Z \ldots$. We use I to denote an index term, for example $i + j$, S to denote a symbol term, for example $\sigma\sigma\sigma\mathbf{0}$ and T to denote string terms. Finally, we use Greek letters $\alpha, \beta, \gamma, \ldots$, to denote formulas.

Definition 14. *$\mathcal{L}_{\mathcal{S}}$-Terms are defined by structural induction as follows:*

1. *Every index variable is a term of type index (index term).*
2. *Every symbol variable is a term of type symbol (symbol term).*
3. *Every string variable is a term of type string (string term).*
4. *If I_1, I_2 are index terms, then so are $(I_1 \circ I_2)$ where $\circ \in \{+, -, \cdot\}$, and* $\mathrm{div}(I_1, I_2)$, $\mathrm{rem}(I_1, I_2)$.
5. *If S is a symbol term then so is σS.*
6. *If T is a string term, then $|T|$ is an index term.*
7. *If I is an index term, and T is a string term, then $e(T, I)$ is a symbol term.*
8. *All constant functions $(0_{\mathrm{index}}, 1_{\mathrm{index}}, \mathbf{0}_{\mathrm{symbol}})$ are terms.*

We employ the lambda operator λ for building terms of type string as we want our theory to be constructive, and have a method for constructing bigger strings from smaller ones. Also, note that a string $\boldsymbol{w}$ of length n, is indexed from zero to $n - 1$

in this chapter as opposed to the previous chapters, where we indexed symbols of $\boldsymbol{w}$ starting from one to n.

Definition 15 (Bound and Free variables)**.** *An occurrence of a variable X in a formula α is said to be bound iff it is in a subformula β of α of the form $\forall X\beta$ or $\exists X\beta$. Otherwise the occurrence is free.*

Definition 16 (Substitution)**.** *Let t_1, t_2 be terms, and α a formula. Then $t_2(t_1/X)$ is the result of replacing all occurrences of X in t_2 by t_1, and $\alpha(t_1/X)$ is the result of replacing all free occurrences of X in α by t_1.*

Definition 17 (String Constructor)**.** *Given a term I of type index, and given a term S of type symbol, then the following is a term T of type string:*

$$\lambda i\langle I, S\rangle. \tag{4.1}$$

T is a term of type string, and is of length I and i occurs in S. The j-th symbol of the string T is obtained by evaluating S at j, i.e., by evaluating $S(j/i)$. Note that, $S(j/i)$ is the term obtained by replacing every *free* occurrence of i in S with j. Since (4.1) is a λ-term, i is considered to be a bound variable and its value ranges from $[0..I-1]$. This concept is formalized in axiom B22. For examples of string constructors see Section 4.3.4.

Definition 18. *$\mathcal{L}_{\mathrm{S}}$-Formulas are defined by structural induction as follows:*

1. *If I_1, I_2 are two index terms, then $I_1 < I_2$ and $I_1 = I_2$ are atomic formulas.*
2. *If S_1, S_2 are symbol terms, then $S_1 < S_2$ and $S_1 = S_2$ are atomic formulas.*
3. *If T_1, T_2 are two string terms, then $T_1 = T_2$ is an atomic formula.*

4. If α, β are formulas (atomic or not), the following are also formulas:

$$\neg\alpha, (\alpha \wedge \beta), (\alpha \vee \beta), \forall X\alpha, \exists X\alpha,$$

where X is a term.

We say that an index quantifier is bounded if it is of the form $\exists i \leq I$ or $\forall i \leq I$, where I is a term of type index and i does not occur free in I. Similarly, we say that a string quantifier is bounded if it is of the form $\exists \boldsymbol{u} \leq I$ or $\forall \boldsymbol{u} \leq I$, where I is an index term, and means that $|\boldsymbol{u}| \leq I$ and $\boldsymbol{u}$ does not occur in I.

Definition 19. *Let Σ_0^B be the set of $\mathcal{L}_{\mathrm{S}}$-formulas without string or symbol quantifiers, where all index quantifiers (if any) are bounded. For $i > 0$, let Σ_i^B (Π_i^B) be the set of $\mathcal{L}_{\mathrm{S}}$ formulas of the form: once the formula is put in prenex form, there are i alternations of bounded string quantifiers, starting with an existential (universal) one, and followed by a Σ_0^B formula.*

Given a formula α, and two terms S_1, S_2 of type symbol, $\mathrm{cond}(\alpha, S_1, S_2)$ is a term of type symbol. We want our theory to be strong enough to prove interesting theorems, but not too strong so that proofs yield feasible algorithms. When a theory is strong (say the axioms express strong properties, such as correctness of $f(\alpha) = t$, where f is a function computing a satisfying assignment t for formula α, when it exists), then the function witnessing the existential quantifiers will be of a higher complexity. For this reason we will restrict the α in the $\mathrm{cond}(\alpha, S_1, S_2)$ to be Σ_0^B. Thus, given such an α and assignments of values to its free variables, we can evaluate the truth value of α, and output the appropriate S_i, in polytime – see Lemma 22.

The alphabet symbols are as follows, $\mathbf{0}$, $\sigma\mathbf{0}$, $\sigma\sigma\mathbf{0}$, $\sigma\sigma\sigma\mathbf{0}$, ..., that is, the unary

function σ allows us to generate as many alphabet symbols as necessary. We are going to abbreviate these symbols as $\sigma_0, \sigma_1, \sigma_2, \sigma_3, \ldots$. In a given application in Stringology, an alphabet of size three would be given by $\Sigma = \{\sigma_0, \sigma_1, \sigma_2\}$, where $\sigma_0 < \sigma_1 < \sigma_2$, inducing a standard lexicographic ordering. We make a point of having an alphabet of any size in the language, rather than a fixed constant size alphabet, as this allows us to formalize arguments of the type: given a particular structure describing strings, show that such strings require alphabets of a given size (see [Buss and Soltys, 2013]).

4.3.3 Semantics of $\mathcal{L}_{\mathcal{S}}$

We denote a structure for $\mathcal{L}_{\mathcal{S}}$ with $\mathcal{M}$. A structure is a way of assigning values to the terms, and truth values to the formulas. We base our presentation of the semantics of $\mathcal{L}_{\mathcal{S}}$ on [Cook and Nguyen, 2010, §II.2.2]. We start with a non-empty set M called the universe. The variables in any $\mathcal{L}_{\mathcal{S}}$ are intended to range over M. Since our theory is three sorted, the universe $M = (\mathcal{I}, \Sigma, \mathcal{S})$, where $\mathcal{I}$ denotes the set of indices, Σ the set of alphabet symbols, and $\mathcal{S}$ the set of strings.

We start by defining the semantics for the three 0-ary (constant) function symbols:

$$0^{\mathcal{M}}_{\text{index}} \in \mathcal{I}, \quad 1^{\mathcal{M}}_{\text{index}} \in \mathcal{I}, \quad 0^{\mathcal{M}}_{\text{symbol}} \in \Sigma,$$

for the two unary function symbol:

$$\sigma^{\mathcal{M}}_{\text{symbol}} : \Sigma \longrightarrow \Sigma, \quad ||^{\mathcal{M}}_{\text{string}} : \mathcal{S} \longrightarrow \mathcal{I},$$

for the six binary function symbols:

$$+^{\mathcal{M}}_{\text{index}} : \mathcal{I}^2 \longrightarrow \mathcal{I}, \quad -^{\mathcal{M}}_{\text{index}} : \mathcal{I}^2 \longrightarrow \mathcal{I}, \quad \cdot^{\mathcal{M}}_{\text{index}} : \mathcal{I}^2 \longrightarrow \mathcal{I}$$

$$\text{div}^{\mathcal{M}}_{\text{index}} : \mathcal{I}^2 \longrightarrow \mathcal{I}, \quad \text{rem}^{\mathcal{M}}_{\text{index}} : \mathcal{I}^2 \longrightarrow \mathcal{I}, \quad e^{\mathcal{M}}_{\text{string}} : \mathcal{S} \times \mathcal{I} \longrightarrow \Sigma.$$

With the function symbols defined according to $\mathcal{M}$, we now associate relations with the predicate symbols, starting with the five binary predicates:

$$<^{\mathcal{M}}_{\text{index}} \subseteq \mathcal{I}^2, \quad =^{\mathcal{M}}_{\text{index}} \subseteq \mathcal{I}^2, \quad <^{\mathcal{M}}_{\text{symbol}} \subseteq \Sigma^2, \quad =^{\mathcal{M}}_{\text{symbol}} \subseteq \Sigma^2, \quad =^{\mathcal{M}}_{\text{string}} \subseteq \mathcal{S}^2,$$

and finally we define the conditional function as follows: $\text{cond}^{\mathcal{M}}_{\text{symbol}}(\alpha, S_1, S_2)$ evaluates to $S_1^{\mathcal{M}}$ if $\alpha^{\mathcal{M}}$ is true, and to $S_2^{\mathcal{M}}$ otherwise. Note that $=^{\mathcal{M}}$ must always evaluate to true equality for all types; that is, equality is hardwired to always be equality. However, all other function symbols and predicates can be evaluated in an arbitrary way (that respects the given arities).

Definition 20. *An* object assignment τ *for a structure* $\mathcal{M}$ *is a mapping from variables to the universe* $M = (\mathcal{I}, \Sigma, \mathcal{S})$*, that is,* M *consists of three sets that we call* indices*,* alphabet symbols*, and* strings*.*

The three sorts are related to each other in that $\mathcal{S}$ can be seen as a function from $\mathcal{I}$ to Σ, i.e., a given $\boldsymbol{u} \in \mathcal{S}$ is just a function $\boldsymbol{u} : \mathcal{I} \longrightarrow \Sigma$. In Stringology we are interested in the case where a given $\boldsymbol{u}$ may be arbitrarily long but it maps $\mathcal{I}$ to a relatively small set of Σ: for example, binary strings map into $\{0, 1\} \subset \Sigma$. Since the range of $\boldsymbol{u}$ is relatively small this leads to interesting structural questions about the mapping: repetitions and patterns.

We start by defining τ on terms: $\tau^{\mathcal{M}}[\sigma]$. Note that if $m \in M$ and X is a variable, then $\tau(m/X)$ denotes the object assignment τ but where we specify that the variable X must evaluate to m.

We define the *evaluation of a term* under $\mathcal{M}$ and τ, by structural induction on the definition of terms given in Section 4.3.1. Suppose I is an index term, we denote its evaluation as $I^{\mathcal{M}}[\tau]$. First, $X^{\mathcal{M}}[\tau]$ is just $\tau(X)$, for each variable X. We must now define object assignments for all the functions. Recall that I, I_1, I_2 are index terms, S is a symbol term and T is a string term.

$$(I_1 \circ_{\text{index}} I_2)^{\mathcal{M}}[\tau] = (I_1^{\mathcal{M}}[\tau] \circ_{\text{index}}^{\mathcal{M}} I_2^{\mathcal{M}}[\tau]),$$

where $\circ \in \{+, -, \cdot\}$ and

$$(\text{div}(I_1, I_2))^{\mathcal{M}}[\tau] = \text{div}^{\mathcal{M}}(I_1^{\mathcal{M}}[\tau], I_2^{\mathcal{M}}[\tau]),$$

$$(\text{rem}(I_1, I_2))^{\mathcal{M}}[\tau] = \text{rem}^{\mathcal{M}}(I_1^{\mathcal{M}}[\tau], I_2^{\mathcal{M}}[\tau]).$$

and for symbol terms we have:

$$(\sigma S)^{\mathcal{M}}[\tau] = \sigma^{\mathcal{M}}(S^{\mathcal{M}}[\tau]).$$

Finally, for string terms:

$$|\text{T}|^{\mathcal{M}}[\tau] = |(\text{T}^{\mathcal{M}}[\tau])|.$$

$$(\text{e}(T, I))^{\mathcal{M}}[\tau] = \text{e}^{\mathcal{M}}(T^{\mathcal{M}}[\tau], I^{\mathcal{M}}[\tau]).$$

Given a formula α, the notation $\mathcal{M} \vDash \alpha[\tau]$, which we read as "$\mathcal{M}$ satisfies α under τ" is also defined by structural induction. We start with the basis case:

$$\mathcal{M} \vDash (S_1 <_{\text{symbol}} S_2)[\tau] \iff (S_1^{\mathcal{M}}[\tau], S_2^{\mathcal{M}}[\tau]) \in <_{\text{symbol}}^{\mathcal{M}} .$$

We deal with the other atomic predicates in a similar way:

$$\mathcal{M} \vDash (I_1 <_{\text{index}} I_2)[\tau] \iff (I_1^{\mathcal{M}}[\tau], I_2^{\mathcal{M}}[\tau]) \in <_{\text{index}}^{\mathcal{M}},$$

$$\mathcal{M} \vDash (I_1 =_{\text{index}} I_2)[\tau] \iff I_1^{\mathcal{M}}[\tau] = I_2^{\mathcal{M}}[\tau],$$

$$\mathcal{M} \vDash (S_1 =_{\text{symbol}} S_2)[\tau] \iff S_1^{\mathcal{M}}[\tau] = S_2^{\mathcal{M}}[\tau],$$

$$\mathcal{M} \vDash (T_1 =_{\text{string}} T_2)[\tau] \iff T_1^{\mathcal{M}}[\tau] = T_2^{\mathcal{M}}[\tau].$$

Now we deal with Boolean connectives:

$$\mathcal{M} \vdash (\alpha \wedge \beta)[\tau] \iff \mathcal{M} \vDash \alpha[\tau] \text{ and } \mathcal{M} \vDash \beta[\tau],$$

$$\mathcal{M} \vdash \neg\alpha[\tau] \iff \mathcal{M} \nvDash \alpha[\tau],$$

$$\mathcal{M} \vdash (\alpha \vee \beta)[\tau] \iff \mathcal{M} \vDash \alpha[\tau] \text{ or } \mathcal{M} \vDash \beta[\tau].$$

Finally, we show how to deal with quantifiers, where the object assignment τ plays a crucial role:

$$\mathcal{M} \vDash (\exists X\alpha)[\tau] \iff \mathcal{M} \vDash \alpha[\tau(m/X)] \text{ for some } m \in M,$$

$$\mathcal{M} \vDash (\forall X\alpha)[\tau] \iff \mathcal{M} \vDash \alpha[\tau(m/X)] \text{ for all } m \in M.$$

Definition 21. *Let* $\underline{\mathbb{S}} = (\mathbb{N}, \Sigma, \mathcal{S})$ *denote the standard model for strings, where* $\mathbb{N} = \{0, 1, 2, \ldots\}$ *is the standard set of natural numbers,* $\Sigma = \{\sigma_0, \sigma_1, \sigma_2, \ldots\}$ *where the alphabet symbols are the ordered sequence* $\sigma_0 < \sigma_1 < \sigma_2, \ldots$, *and where* $\mathcal{S}$ *is the set of functions* $\boldsymbol{u} : \mathcal{I} \longrightarrow \Sigma$, *and where all the function and predicate symbols get their standard interpretations.*

Lemma 22. *Given any formula* $\alpha \in \Sigma_0^B$, *and a particular object assignment* τ, *we can verify* $\underline{\mathbb{S}} \models \alpha[\tau]$ *in polytime in the lengths of the strings and values of the indices in* α.

Proof. Let t, t_1, t_2 denote terms of any type (index, symbol or strings). We first show that evaluating a term t, i.e., computing $t^{\underline{\mathbb{S}}}[\tau]$, can be done in polytime. We do this by structural induction on t. If t is just a variable then there are three cases:

- t is equal to i an index variable. Therefore $i^{\underline{\mathbb{S}}}[\tau] = \tau(i) \in \mathbb{N}$.
- t is equal to a a symbol variable. Therefore, $a^{\underline{\mathbb{S}}}[\tau] = \tau(a) \in \Sigma$.
- t is equal to $\boldsymbol{u}$ a string variable. Therefore, $\boldsymbol{u}^{\underline{\mathbb{S}}}[\tau] = \tau(\boldsymbol{u}) \in S$.

Note that the assumption is that computing $\tau(X)$ is for free, as τ is given as a table which states which free variable gets replaced by what concrete value. Therefore evaluating t in the base case when it is just a variable is done in constant time.

When t is a term involving function symbols. Recall that all index values are assumed to be given in unary, and all the function operations we have are clearly polytime in the values of the arguments (index addition, subtraction, multiplication, etc.). Therefore, evaluating t in this case is done in polytime in the values of the arguments. We therefore conclude that evaluating any term t can be done in polytime.

Consider evaluating the atomic formulas $(t_1 < t_2)^{\underline{\mathbb{S}}}[\tau]$ and $(t_1 = t_2)^{\underline{\mathbb{S}}}[\tau]$, where t_1 and t_2 are terms of any type (index, symbols or strings). We already established that $t_1^{\underline{\mathbb{S}}}[\tau]$ and $t_2^{\underline{\mathbb{S}}}[\tau]$ can be computed in polytime. Comparing integers can be done in polytime, and the same holds for other atomic formulas. Therefore, evaluating atomic formulas can be done in polytime. Now, we consider evaluating formulas with Boolean connectives $\wedge, \vee$. Since performing these Boolean operations can be done in polytime, we conclude that evaluating formulas with Boolean connectives can be done in polytime.

Finally, we consider quantification; but we are only allowed bounded index quantification: $(\exists i \leq t\alpha)^{\underline{\mathbb{S}}}[\tau]$, and $(\exists i \leq t\alpha)^{\underline{\mathbb{S}}}[\tau]$. This is equivalent to computing:

$$\bigvee_{j=0}^{t^{\underline{\mathbb{S}}}[\tau]} \alpha^{\underline{\mathbb{S}}}[\tau(j/i)], \text{ and } \bigwedge_{j=0}^{t^{\underline{\mathbb{S}}}[\tau]} \alpha^{\underline{\mathbb{S}}}[\tau(j/i)].$$

Since this can be done in polytime, we conclude that evaluating any formula can be done in polytime. □

The point of Section 4.3 is to show that our theory is completely constructive, that is, our strings can be built from the ground up. For example, we cannot build a string with the lambda constructor that encodes all the Turing machines that halt on all input, though such a string exists. This is the point of this meticulous theory; to have a robust foundation for the field of Stringology. Stringologists study strings, but where do they come from? We show where they come from.

4.3.4 Examples of string constructors

The string 000 can be represented by:

$$\lambda i\langle 1+1+1, \mathbf{0}\rangle.$$

Given an integer n, let $\hat{n}$ abbreviate the term $1+1+\cdots+1$ consisting of n many 1s. Using this convenient notation, a string of length 8 of alternating 1s and 0s can be represented by:

$$\lambda i\langle \hat{8}, \mathrm{cond}(\exists j \leq i(j+j=i), \mathbf{0}, \sigma\mathbf{0})\rangle. \tag{4.2}$$

Note that this example illustrates that indices are going to be effectively encoded in unary; this is fine as we are proposing a theory for strings, and so unary indices are an encoding that is linear in the length of the string. The same point is made in [Cook and Nguyen, 2010], where the indices are assumed to be encoded in unary, because the main object under investigation are binary strings, and the complexity is measured in the lengths of the strings, and unary encoded indices are proportional to those lengths.

Also note that there are various ways to represent the same string; for example, the string given by (4.2) can also be written as:

$$\lambda i\langle \hat{2}\cdot\hat{4}, \mathrm{cond}(\exists j \leq i(j+j=i+1), \sigma\mathbf{0}, \mathbf{0})\rangle. \tag{4.3}$$

For convenience, we define the empty string ε as follows:

$$\varepsilon := \lambda i\langle 0, \mathbf{0}\rangle.$$

Let $\boldsymbol{u}$ be a binary string, and suppose that we want to define $\bar{\boldsymbol{u}}$, which is $\boldsymbol{u}$ with every 0 (denoted $\mathbf{0}$) flipped to 1 (denote $\sigma\mathbf{0}$), and every 1 flipped to 0. We can define $\bar{\boldsymbol{u}}$ as follows:

$$\bar{\boldsymbol{u}} := \lambda i\langle |\boldsymbol{u}|, \mathrm{cond}(e(\boldsymbol{u}, i) = \mathbf{0}, \sigma\mathbf{0}, \mathbf{0}\rangle.$$

We can also define a string according to properties of positions of indices; suppose we wish to define a binary string of length n which has one in all positions which are multiples of 3:

$$\boldsymbol{v} := \lambda i\langle \hat{n}, \mathrm{cond}(\exists j \leq n(i = j + j + j), \sigma\mathbf{0}, \mathbf{0})\rangle.$$

Note that both $\bar{\boldsymbol{u}}$ and $\boldsymbol{v}$ are defined with the conditional function where the formula α conforms to the restriction: variables are either free (like $\boldsymbol{u}$ in $\bar{\boldsymbol{u}}$), or, if quantified, all such variables are bounded and of type index (like j in $\boldsymbol{v}$).

Note that given a string $\boldsymbol{w}$, $|\boldsymbol{w}|$ is its length. However, we number the positions of a string starting at zero, and hence the last position is $|\boldsymbol{w}| - 1$. For $j \geq |\boldsymbol{w}|$ we are going to define a string to be just $\mathbf{0}$s.

Suppose we want to define the reverse of a string, namely if $\boldsymbol{u} = \boldsymbol{u}_0\boldsymbol{u}_1 \ldots \boldsymbol{u}_{n-1}$, then its reverse is $\boldsymbol{u}^R = \boldsymbol{u}_{n-1}\boldsymbol{u}_{n-2} \ldots \boldsymbol{u}_0$. Then,

$$\boldsymbol{u}^R := \lambda i\langle |\boldsymbol{u}|, e(\boldsymbol{u}, (|\boldsymbol{u}| - 1) - i)\rangle,$$

and the concatenation of two strings, which we denote as "$\cdot$", can be represented as follows:

$$\boldsymbol{u} \cdot \boldsymbol{v} := \lambda i\langle |\boldsymbol{u}| + |\boldsymbol{v}|, \mathrm{cond}(i < |\boldsymbol{u}|, e(\boldsymbol{u}, i), e(\boldsymbol{v}, i - |\boldsymbol{u}|))\rangle. \tag{4.4}$$

4.3.5 Axioms of the theory $\mathcal{S}$

We assume that we have the standard equality axioms (defining that equality is an congruence relation with respect to the elements, function and predicate symbols in the language) which assert that equality is true equality — see [Buss, 1998, §2.2.1], and therefore we don't give those axioms explicitly.

Since we are going to use the rules of Gentzen's calculus, LK, we present the axioms as Gentzen's sequents, that is, they are of the form $\Gamma \to \Delta$, where Γ, Δ are coma-separated lists of formulas. Specifically, the sequent is of the form:

$$\alpha_1, \alpha_2, \ldots, \alpha_n \to \beta_1, \beta_2, \ldots, \beta_m,$$

where n or m (or both) may be zero, that is, Γ or Δ (or both) may be empty. The semantics of sequents is as follows: a sequent is valid if for any structure $\mathcal{M}$ that satisfies all the formulas in Γ, satisfies at least one formula in Δ. Using the standard Boolean connectives this can be state as follows: $\neg \bigwedge_i \alpha_i \vee \bigvee_j \beta_j$, where $1 \leq i \leq n$ and $1 \leq j \leq m$. To prove a formula α we need to derive $\to \alpha$ and to refute it we need to derive $\alpha \to$.

The index axioms are the same as 2-BASIC in [Cook and Nguyen, 2010, pg. 96], plus we add four more axioms (B7 and B15, B8 and B16) to define bounded subtraction, as well as division and remainder functions. The 2-BASIC axioms: $B1, B2$, defines that there are no negative numbers and that the successor function is a bijection. The axioms $B3, B4$, and $B5, B6$ provide recursive definitions of addition and multiplication respectively. Axioms $B9, B10$ provide basic properties of $\leq$ and the axioms $B12$ expresses that the elements of the index sort are totally ordered. Finally

the axiom B11 defines that 0 is the minimum number, B13 defines discreteness, B14 defines the predecessor function.

A formula α is equivalent to a sequent $\rightarrow \alpha$, and so, for readability we sometimes mix the two.

Index Axioms	
B1. $i+1 \neq 0$	B9. $i \leq j, j \leq i \rightarrow i = j$
B2. $i+1 = j+1 \rightarrow i = j$	B10. $i \leq i+j$
B3. $i+0 = i$	B11. $0 \leq i$
B4. $i+(j+1) = (i+j)+1$	B12. $i \leq j \vee j \leq i$
B5. $i \cdot 0 = 0$	B13. $i \leq j \leftrightarrow i < j+1$
B6. $i \cdot (j+1) = (i \cdot j)+i$	B14. $i \neq 0 \rightarrow \exists j \leq i(j+1 = i)$
B7. $i \leq j, i+k = j \rightarrow j-i = k$	B15. $i \not\leq j \rightarrow j-i = 0$
B8. $j \neq 0 \rightarrow \mathrm{rem}(i,j) < j$	B16. $j \neq 0 \rightarrow i = j \cdot \mathrm{div}(i,j) + \mathrm{rem}(i,j)$

The alphabet axioms express that the alphabet is totally ordered according to "$<$" and define the function cond.

Alphabet Axioms
B17. $a \lneqq \sigma a$
B18. $a < b, b < c \rightarrow a < c$
B19. $\alpha \rightarrow \mathrm{cond}(\alpha, a, b) = a$
B20. $\neg\alpha \rightarrow \mathrm{cond}(\alpha, a, b) = b$

Note that α in cond is a formula with the following restrictions: it only allows bounded index quantifiers and hence evaluates to true or false once all free variables have been assigned values. Hence cond always yields the symbol term S_1 or the symbol term S_2, according to the truth value of α.

Note that the alphabet symbol type is defined by four axioms, B17–B20, two of which define the cond function. These four axioms define symbols to be ordered "place holders" and nothing more. This is consistent with alphabet symbols in classical Stringology, where there are no operations defined on them (for example, we do not add or multiply alphabet symbols).

Finally, these are the axioms governing strings:

String Axioms
B21. $\lvert\lambda i\langle I, S\rangle\rvert = I$
B22. $j < I \rightarrow e(\lambda i\langle I, S\rangle, j) = S(j/i)$
B23. $\lvert\boldsymbol{u}\rvert \leq j \rightarrow e(\boldsymbol{u}, j) = \mathbf{0}$
B24. $\lvert\boldsymbol{u}\rvert = \lvert\boldsymbol{v}\rvert, \forall i < \lvert\boldsymbol{u}\rvert e(\boldsymbol{u}, i) = e(\boldsymbol{v}, i) \rightarrow \boldsymbol{u} = \boldsymbol{v}$

Note that axioms B22–24 define the structure of a string. In our theory, a string can be given as a variable, or it can be constructed. Axiom B21 defines the length of the constructed strings, and axiom B22 shows that if j is less than the length of the string, then the symbol in position j is given by substituting j for all the free occurrences of i in S; this is the meaning of $S(j/i)$. On the other hand, B23 says that if j is greater or equal to the length of a string, then $e(\boldsymbol{u}, j)$ defaults to $\mathbf{0}$. The last axioms, B24, says that if two strings $\boldsymbol{u}$ and $\boldsymbol{v}$ have the same length, and the corresponding symbols are equal, then the two strings are in fact equal.

In axiom B24 there are three types of equalities, from left to right: index, symbol, and string, and so B24 is the axiom that ties all three sorts together. Note that formally strings are infinite ordered sequences of alphabet symbols. But we conclude that they are equal based on comparing finitely many entries ($\forall i < \lvert\boldsymbol{u}\rvert e(\boldsymbol{u}, i) = e(\boldsymbol{v}, i)$). This works because by B23 we know that for $i \geq \lvert\boldsymbol{u}\rvert$, $e(\boldsymbol{u}, i) = e(\boldsymbol{v}, i) = \mathbf{0}$

(since $|\boldsymbol{u}| = |\boldsymbol{v}|$ by the assumption in the antecedent). A standard string of length n is an object of the form:

$$\sigma_{i_0}, \sigma_{i_1}, \ldots, \sigma_{i_{n-1}}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \ldots,$$

i.e., an infinite string indexed by the natural numbers, where there is a position so that all the elements greater than that position are $\mathbf{0}$.

4.3.6 The rules of $\mathcal{S}$

We use the Gentzen's predicate calculus, LK, as presented in [Buss, 1998].

Weak structural rules

exchange-left: $\dfrac{\Gamma_1, \alpha, \beta, \Gamma_2 \to \Delta}{\Gamma_1, \beta, \alpha, \Gamma_2 \to \Delta}$ exchange-right: $\dfrac{\Gamma \to \Delta_1, \alpha, \beta, \Delta_2}{\Gamma \to \Delta_1, \beta, \alpha, \Delta_2}$

contraction-left: $\dfrac{\alpha, \alpha, \Gamma \to \Delta}{\alpha, \Gamma \to \Delta}$ contraction-right: $\dfrac{\Gamma \to \Delta, \alpha, \alpha}{\Gamma \to \Delta, \alpha}$

weakening-left: $\dfrac{\Gamma \to \Delta}{\alpha, \Gamma \to \Delta}$ weakening-right: $\dfrac{\Gamma \to \Delta}{\Gamma \to \Delta, \alpha}$

Cut rule

$$\frac{\Gamma \to \Delta, \alpha \quad \alpha, \Gamma \to \Delta}{\Gamma \to \Delta}$$

Rules for introducing connectives

$\neg$-left: $\dfrac{\Gamma \to \Delta, \alpha}{\neg\alpha, \Gamma \to \Delta}$ $\neg$-right: $\dfrac{\alpha, \Gamma \to \Delta}{\Gamma \to \Delta, \neg\alpha}$

$\wedge$-left: $\dfrac{\alpha, \beta, \Gamma \to \Delta}{\alpha \wedge \beta, \Gamma \to \Delta}$ $\wedge$-right: $\dfrac{\Gamma \to \Delta, \alpha \quad \Gamma \to \Delta, \beta}{\Gamma \to \Delta, \alpha \wedge \beta}$

$\vee$-left: $\dfrac{\alpha, \Gamma \to \Delta \quad \beta, \Gamma \to \Delta}{\alpha \vee \beta, \Gamma \to \Delta}$ $\vee$-right: $\dfrac{\Gamma \to \Delta, \alpha, \beta}{\Gamma \to \Delta, \alpha \vee \beta}$

Rules for introducing quantifiers

$\forall$-left: $\dfrac{\alpha(t), \Gamma \to \Delta}{\forall X \alpha(X), \Gamma \to \Delta}$ $\forall$-right: $\dfrac{\Gamma \to \Delta, \alpha(b)}{\Gamma \to \Delta, \forall X \alpha(X)}$

$\exists$-left: $\dfrac{\alpha(b), \Gamma \to \Delta}{\exists X \alpha(X), \Gamma \to \Delta}$ $\exists$-right: $\dfrac{\Gamma \to \Delta, \alpha(t)}{\Gamma \to \Delta, \exists X \alpha(X)}$

Note that b must be free in Γ, Δ.

Induction rule

$$\text{Ind:} \quad \frac{\Gamma, \alpha(i) \to \alpha(i+1), \Delta}{\Gamma, \alpha(0) \to \alpha(I), \Delta}$$

where i does not occur free in Γ, Δ, and I is a term of type index. By restricting the quantifier structure of α, we control the strength of this induction. We call Σ^B_i-Ind to be the induction rule where α is restricted to be in Σ^B_i. We are mainly interested in Σ^B_i-Ind where $i = 0$ or $i = 1$, as otherwise, it might not be possible to compute the actual value of the existential quantifier feasibly.

Definition 23. *Let $\mathcal{S}_i$ to be the set of formulas (sequents) derivable from the axioms B1-24 using the rules of LK, where the α formula in* cond *is restricted to be in Σ^B_0 and where we use Σ^B_i-Ind.*

Theorem 24 (Cut-Elimination)**.** *If Φ is a $\mathcal{S}_i$ proof of a formula α, then Φ can always be converted into a Φ' $\mathcal{S}_i$ proof where the cut rule is applied only to formulas in Σ^B_i.*

We do not prove Theorem 24, as it is based on the same type of reasoning given in [Soltys, 1999].

The point of the Cut-Elimination Theorem is that in any $\mathcal{S}_i$ proof we can always limit all the intermediate formulas to be in Σ_i^B, i.e., we do not need to construct intermediate formulas whose quantifier complexity (number of alternations of $\forall$ and $\exists$ quantifiers) is more than that of the conclusion.

As an example of the use of $\mathcal{S}_i$ we outline an $\mathcal{S}_0$ proof of the equality of (4.2) and (4.3). First note that by axiom B21 we have that:

$$|\lambda i\langle\hat{8}, \mathrm{cond}(\exists j \leq i(j + j = i), \mathbf{0}, \sigma\mathbf{0})\rangle| = \hat{8}$$
$$|\lambda i\langle\hat{2} \cdot \hat{4}, \mathrm{cond}(\exists j \leq i(j + j = i + 1), \sigma\mathbf{0}, \mathbf{0})\rangle| = \hat{2} \cdot \hat{4},$$

and by axioms B1-16 we can prove that $\hat{8} = \hat{2} \cdot \hat{4}$ as follows,

$$\begin{aligned}
\hat{2} \cdot \hat{4} &= \hat{2} \cdot (\hat{3} + 1) \\
&= (\hat{2} \cdot \hat{3}) + \hat{2} && (\Leftarrow \text{from Axiom B6}) \\
&= \hat{2} \cdot (\hat{2} + 1) + \hat{2} \\
&= (\hat{2} \cdot \hat{2}) + \hat{2} + \hat{2} && (\Leftarrow \text{from Axiom B6}) \\
&= \hat{2} \cdot (1 + 1) + \hat{2} + \hat{2} \\
&= (\hat{2} \cdot 1) + \hat{2} + \hat{2} + \hat{2} && (\Leftarrow \text{from Axiom B6}) \\
&= \hat{2} \cdot (0 + 1) + \hat{2} + \hat{2} + \hat{2} \\
&= (\hat{2} \cdot 0) + \hat{2} + \hat{2} + \hat{2} + \hat{2} && (\Leftarrow \text{from Axiom B6}) \\
&= 0 + \hat{2} + \hat{2} + \hat{2} + \hat{2} && (\Leftarrow \text{from Axiom B5}) \\
&= \hat{2} + \hat{2} + \hat{2} + \hat{2} && (\Leftarrow \text{from Axiom B3 and} \\
& && \text{commutative property of } +)
\end{aligned}$$

$$= 1+1+1+1+1+1+1+1 \qquad (\Leftarrow \text{Expansion of } \hat{2})$$

Since, $\hat{8} = 1+1+1+1+1+1+1+1$, we can conclude by transitivity of equality (equality is always true equality) that $\hat{8} = \hat{2} \cdot \hat{4}$ and :

$$|\lambda i\langle\hat{8}, \text{cond}(\exists j \leq i(j+j=i), \mathbf{0}, \sigma\mathbf{0})\rangle| = |\lambda i\langle\hat{2}\cdot\hat{4}, \text{cond}(\exists j \leq i(j+j=i+1), \sigma\mathbf{0}, \mathbf{0})\rangle|.$$

Now we have to show that:

$$\forall i < \hat{8}(\text{cond}(\exists j \leq i(j+j=i), \mathbf{0}, \sigma\mathbf{0}) = \text{cond}(\exists j \leq i(j+j=i+1), \sigma\mathbf{0}, \mathbf{0})) \quad (4.5)$$

and then, using axiom B24 and some cuts on Σ_0^B formulas we can prove that in fact the two terms given by (4.2) and (4.3) are equal.

In order to prove (4.5) we show that:

$$i < \hat{8} \wedge (\text{cond}(\exists j \leq i(j+j=i), \mathbf{0}, \sigma\mathbf{0}) = \text{cond}(\exists j \leq i(j+j=i+1), \sigma\mathbf{0}, \mathbf{0})) \quad (4.6)$$

and then we can introduce the quantifier with $\forall$-intro right. We prove (4.6) by proving:

$$i < \hat{8} \rightarrow \text{cond}(\exists j \leq i(j+j=i), \mathbf{0}, \sigma\mathbf{0}) = \text{cond}(\exists j \leq i(j+j=i+1), \sigma\mathbf{0}, \mathbf{0}) \quad (4.7)$$

Now to prove (4.7) we have to show that:

$$\mathcal{S}_0 \vdash \exists j \leq i(j+j=i) \leftrightarrow \neg\exists j \leq i(j+j=i+1),$$

Then, using B19 and B20 we can show (4.7).

If we prove a formula in a theory, say $\exists \boldsymbol{u} \leq I\alpha(\boldsymbol{u}, \boldsymbol{v})$, then the function f which on input $\boldsymbol{v}$ outputs $\boldsymbol{u}$, that is $f(\boldsymbol{u}) = \boldsymbol{v}$, is called the *witnessing function*.

4.4 Witnessing theorem for $\mathcal{S}$

Recall that $\mathcal{S}_1$ is our string theory restricted to Σ_1^B-Ind. For convenience, we sometimes use the notation $\vec{\boldsymbol{v}}$, to denote several string variables, i.e., $\vec{\boldsymbol{v}} = \boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_\ell$.

We now prove the main theorem for this chapter, showing that if we manage to prove in $\mathcal{S}_1$ the existence of a string $\boldsymbol{u}$ with some given properties, then in fact we can construct such a string with a polytime algorithm.

Theorem 25 (Witnessing)**.** *If $\mathcal{S}_1 \vdash \exists \vec{\boldsymbol{u}} \leq I\alpha(\boldsymbol{u}, \vec{\boldsymbol{v}})$, then it is possible to compute $\vec{\boldsymbol{u}}$ in polynomial time in the total length of all the string variables in $\vec{\boldsymbol{v}}$ and the value of all the free index variables in α.*

Proof. We give an outline of the proof of the Witnessing theorem. In order to simplify the proof we show it for $\mathcal{S}_1 \vdash \exists \boldsymbol{u} \leq I\alpha(\boldsymbol{u}, \vec{\boldsymbol{v}})$, i.e., $\boldsymbol{u}$ is a single string variable rather than a set, i.e., rather than a block of bounded existential string quantifiers. The general proof is very similar.

We argue by induction on the number of lines in the proof of $\exists \boldsymbol{u} \leq I\alpha(\boldsymbol{u}, \vec{\boldsymbol{v}})$ that $\boldsymbol{u}$ can be witnessed by a polytime algorithm. Each line in the proof is either an axiom (see Section 4.3.5), or follows from previous lines by the application of a rule (see Section 4.3.6). By Theorem 24 we know that all the formulas in the $\mathcal{S}_1$ proof of $\exists \boldsymbol{u} \leq I\alpha(\boldsymbol{u}, \vec{\boldsymbol{v}})$ can be restricted to be Σ_1^B. It is this fundamental application of Cut-Elimination that allows us to prove our Witnessing theorem.

The Basis Case is simple as the axioms have no string quantifiers. In the induction

step the two cases are $\exists$-right and the induction rule. In the former case we have:

$$\exists\text{-right: } \frac{|T| \leq I, \Gamma \rightarrow \Delta, \alpha(T, \vec{\boldsymbol{v}}, \vec{i})}{\Gamma \rightarrow \Delta, \exists \boldsymbol{u} \leq I \alpha(\boldsymbol{u}, \vec{\boldsymbol{v}}, \vec{i})}$$

which is the $\exists$-right rule adapted to the case of bounded string quantification, where T is a string term and I is an index term. We use $\vec{\boldsymbol{v}}$ to denote all the free string variables, and $\vec{i}$ to denote explicitly all the free index variables. Then $\boldsymbol{u}$ is witnessed by the function $f(\vec{\boldsymbol{v}}, \vec{i})$ and when f is evaluated at $\vec{A}$ and $\vec{b}$ we get $\boldsymbol{u}$, that is:

$$f(\vec{A}, \vec{b}) := T^{\underline{\mathbb{S}}}[\tau(\vec{A}/\vec{\boldsymbol{v}})(\vec{b}/\vec{i})].$$

By Lemma 22 we know that we can evaluate any $\mathcal{L}_S$-term in $\underline{\mathbb{S}}$ in polytime in the length of the free string variables and the values of the index variables. Therefore, f is polytime as evaluating T under $\underline{\mathbb{S}}$ and any object assignment can by done in polytime.

For the induction case we restate the rule as follows in order to make all the free variables more explicit:

$$\frac{\boldsymbol{u} \leq I, \alpha(\boldsymbol{u}, \vec{\boldsymbol{v}}, i, \vec{j}) \rightarrow \exists \boldsymbol{u} \leq I \alpha(\boldsymbol{u}, \vec{\boldsymbol{v}}, i+1, \vec{j})}{\boldsymbol{u} \leq I, \alpha(\boldsymbol{u}, \vec{\boldsymbol{v}}, 0, \vec{j}) \rightarrow \exists \boldsymbol{u} \leq I \alpha(\boldsymbol{u}, \vec{\boldsymbol{v}}, I', \vec{j})}$$

where $\vec{j}$ denotes all the free variables and I' is an index term. We ignore Γ, Δ for clarity, and we ignore existential quantifiers on the left side, as it is quantifiers on the right side that we are interested in witnessing. The algorithm is clear: suppose we have a $\boldsymbol{u}$ such that $\alpha(\boldsymbol{u}, \vec{\boldsymbol{v}}, 0, \vec{\boldsymbol{v}})$ is satisfied. Use top of rule to compute $\boldsymbol{u}$'s for $i = 1, 2, \ldots, I^{\underline{\mathbb{S}}}[\tau]$. $\square$

4.5 Application of $\mathcal{S}$ to Stringology

In this section we state some basic Stringology constructions as $\mathcal{L}_{\mathcal{S}}$ formulas.

Prefix, Suffix and Subwords

The prefix, suffix, and subword are basic constructs of a given string $\boldsymbol{v}$. The $\mathcal{L}_{\mathcal{S}}$-term for a prefix of length i is given by $\lambda k\langle i, e(\boldsymbol{v}, k)\rangle$, and a $\mathcal{L}_{\mathcal{S}}$-term for a suffix of length i is given by $\lambda k\langle i, e(\boldsymbol{v}, |\boldsymbol{v}| - i + 1 + k)\rangle$, and since any subword of length i, is a prefix (of the same length) of some suffix of length j, the $\mathcal{L}_{\mathcal{S}}$-term is given by $\lambda l\langle i, e(\lambda k\langle j, e(\boldsymbol{v}, |\boldsymbol{v}| - j + 1 + k)\rangle, l)\rangle$.

We can state that $\boldsymbol{u}$ is a prefix of $\boldsymbol{v}$ with the following Σ_0^B predicate:

$$\mathrm{pre}(\boldsymbol{u}, \boldsymbol{v}) := \exists i \leq |\boldsymbol{v}|(\boldsymbol{u} = \lambda k\langle i, e(\boldsymbol{v}, k)\rangle),$$

The predicate for suffix $\mathrm{suf}(\boldsymbol{u}, \boldsymbol{v})$ is defined similarly and is given by:

$$\mathrm{suf}(\boldsymbol{u}, \boldsymbol{v}) := \exists i \leq |\boldsymbol{v}|(\boldsymbol{u} = \lambda k\langle i, e(\boldsymbol{v}, |\boldsymbol{v}| - i + 1 + k)\rangle),$$

Finally, the predicate for subword $\mathrm{sub}(\boldsymbol{u}, \boldsymbol{v})$ is given by:

$$\mathrm{sub}(\boldsymbol{u}, \boldsymbol{v}) := \exists i, j \leq |\boldsymbol{v}|(\boldsymbol{u} = \lambda l\langle i, e(\lambda k\langle j, e(\boldsymbol{v}, |\boldsymbol{v}| - j + 1 + k)\rangle, l)\rangle)$$

Counting symbols

Suppose that we want to count the number of occurrences of a particular symbol σ_i in a given string $\boldsymbol{u}$; this can be defined with the notation $(\boldsymbol{u})_{\sigma_i}$, but we need to define

this function with a new axiom (as the language given thus far is not suitable for defining $(\boldsymbol{u})_{\sigma_i}$ with a term). First, define the projection of a string $\boldsymbol{u}$ according to σ_i as follows:

$$\boldsymbol{u}|_{\sigma_i} := \lambda k \langle |\boldsymbol{u}|, \mathrm{cond}(e(\boldsymbol{u}, k) = \sigma_i, \sigma_1, \sigma_0) \rangle.$$

That is, $\boldsymbol{u}|_{\sigma_i}$ is effectively a binary string with 1s where $\boldsymbol{u}$ had σ_i, and 0s everywhere else, and of the same length as $\boldsymbol{u}$. Thus, counting σ_i's in $\boldsymbol{u}$ is the same as counting 1's in $\boldsymbol{u}|_{\sigma_i}$. Given a binary string $\boldsymbol{v}$, we define $(\boldsymbol{v})_{\sigma_1}$ as follows:

C1. $|\boldsymbol{v}| = 0 \rightarrow (\boldsymbol{v})_{\sigma_1} = 0$

C2. $|\boldsymbol{v}| \geq 1, e(\boldsymbol{v}, 0) = \sigma_0 \rightarrow (\boldsymbol{v})_{\sigma_1} = (\lambda i \langle |\boldsymbol{v}| - 1, e(\boldsymbol{v}, i+1) \rangle)_{\sigma_1}$

C3. $|\boldsymbol{v}| \geq 1, e(\boldsymbol{v}, 0) = \sigma_1 \rightarrow (\boldsymbol{v})_{\sigma_1} = 1 + (\lambda i \langle |\boldsymbol{v}| - 1, e(\boldsymbol{v}, i+1) \rangle)_{\sigma_1}$

Having defined $(\boldsymbol{u})_{\sigma_1}$ with axioms C1-3, and $\boldsymbol{u}|_{\sigma_i}$ as a term in $\mathcal{L}_{\mathbb{S}}$, we can now define $(\boldsymbol{u})_{\sigma_i}$ as follows: $(\boldsymbol{u}|_{\sigma_i})_{\sigma_1}$. Note that C1-3 are Σ_0^B sequents.

Borders

Let $\mathrm{Brd}(\boldsymbol{v}, i)$ be the border predicate which asserts that the string $\boldsymbol{v}$ has a border (see Section 1.4) of size i. We define the border predicate as:

$$\mathrm{Brd}(\boldsymbol{v}, i) := \lambda k \langle i, e(\boldsymbol{v}, k) \rangle = \lambda k \langle i, e(\boldsymbol{v}, |\boldsymbol{v}| - i + 1 + k) \rangle \wedge i < |\boldsymbol{v}|,$$

$\mathrm{MaxBrd}(\boldsymbol{v}, i)$ is the predicate that states that i is the largest possible border size:

$$\mathrm{MaxBrd}(\boldsymbol{v}, i) := \mathrm{Brd}(\boldsymbol{v}, i) \wedge (\neg \mathrm{Brd}(\boldsymbol{v}, i+1) \vee |\boldsymbol{u}| = |\boldsymbol{v}| - 1).$$

Periodicity and Periodicity Lemma

The Periodicity Lemma states the following: Suppose that p and q are two periods (see Section 1.4) of string $\boldsymbol{v}$, $|\boldsymbol{v}| = n$, and $d = \gcd(p, q)$. If $p + q \leq n + d$, then d is also a period of $\boldsymbol{v}$.

Let $\mathrm{Prd}(\boldsymbol{v}, p)$ be true if p is a period of the string $\boldsymbol{v}$. Note that $\boldsymbol{u}$ is a border of a string $\boldsymbol{v}$ if and only if $p = |\boldsymbol{v}| - |\boldsymbol{u}|$ is a period of $\boldsymbol{v}$. Using this observation we can define the predicate for a period as a Σ_0^B formula:

$$\mathrm{Prd}(\boldsymbol{v}, p) := \exists i < |\boldsymbol{v}|(p = |\boldsymbol{v}| - i \wedge \mathrm{Brd}(\boldsymbol{v}, i))$$

We can state with a Σ_0^B formula that $d = \gcd(i, j)$: $\mathrm{rem}(d, i) = \mathrm{rem}(d, j) = 0$, and $\mathrm{rem}(d', i) = \mathrm{rem}(d', j) = 0 \supset d' \leq d$. We can now state the Periodicity Lemma as the sequent $\mathrm{PL}(\boldsymbol{v}, p, q)$ where all formulas are Σ_0^B as:

$$\mathrm{Prd}(\boldsymbol{v}, p), \mathrm{Prd}(\boldsymbol{v}, q), \exists d \leq p(d = \gcd(p, q) \wedge p + q \leq |\boldsymbol{v}| + d) \rightarrow \mathrm{Prd}(\boldsymbol{v}, d).$$

Chapter 5

Future Directions and Open Problems

We conclude the thesis with future directions related to each chapter followed by a list of open problems in the area.

5.1 Square-free Strings over Alphabet Lists

In the area of avoiding repetitions in strings and studying admissibility of the class $\mathcal{L}_3$ in particular, we present various approaches to proving the conjecture that $\mathcal{L}_3$ is admissible ranging from patterns in strings to Proof Complexity. We finally conclude this section with a list of open problems related to this area.

The work presented in Sections 5.1.1 and 5.1.2 has recently been accepted for publication at the Prague Stringology Conference (PSC) 2016.

5.1.1 Offending Suffix Pattern

In this section, we introduce a pattern that we call an "offending suffix", and we show in Lemma 26 that such suffixes characterize in a meaningful way strings over alphabet lists with squares.

Let $\mathcal{C}(n)$, an *offending suffix*, be a pattern defined recursively:

$$\begin{aligned} \mathcal{C}(1) &= \boldsymbol{X}_1 a_1 \boldsymbol{X}_1, \text{ and for } n > 1, \\ \mathcal{C}(n) &= \boldsymbol{X}_n \mathcal{C}(n-1) a_n \boldsymbol{X}_n \mathcal{C}(n-1). \end{aligned} \tag{5.1}$$

To be more precise, given a morphism, $h : \Delta^* \to \Sigma^*$, we call $h(\{a_1, a_2, \ldots, a_n\}) \subseteq \Sigma$ the pivots of h. When all the variables in the set $\{\boldsymbol{X}_1, \boldsymbol{X}_2, \ldots, \boldsymbol{X}_n\}$ map to ε, we get the pattern for the shortest possible offending suffix for a list $L \in \mathcal{L}_n$. We call this pattern the *shortest offending suffix*, and employ the notation:

$$\mathcal{C}_s(n) = a_1 a_2 a_1 \ldots a_n \ldots a_1 a_2 a_1. \tag{5.2}$$

Note that $|\mathcal{C}_s(n)| = 2|\mathcal{C}_s(n-1)| + 1$, where $|\mathcal{C}_s(1)| = 1$, and so, $|\mathcal{C}_s(n)| = 2^n - 1$.

As we are interested in offending suffixes for $\mathcal{L}_3$, we consider mainly:

$$\begin{aligned} \mathcal{C}(3) &= \boldsymbol{X}_3 \boldsymbol{X}_2 \boldsymbol{X}_1 a_1 \boldsymbol{X}_1 a_2 \boldsymbol{X}_2 \boldsymbol{X}_1 a_1 \boldsymbol{X}_1 a_3 \boldsymbol{X}_3 \boldsymbol{X}_2 \boldsymbol{X}_1 a_1 \boldsymbol{X}_1 a_2 \boldsymbol{X}_2 \boldsymbol{X}_1 a_1 \boldsymbol{X}_1, \\ \mathcal{C}_s(3) &= a_1 a_2 a_1 a_3 a_1 a_2 a_1, \end{aligned} \tag{5.3}$$

and observe that $\mathcal{C}_s(3) a_i$, for $i = 1, 2, 3$, all map to strings with squares.

Pattern $\mathcal{C}$ in (5.1) bears great resemblance to Zimin words (1.2). Comparing (1.2) to (5.1), one can see that mapping $\boldsymbol{X}_i$ to a_i in (1.2) yields the same string as mapping

$\boldsymbol{X}_i$ to ε in (5.1). In particular, the shortest offending suffix $\mathcal{C}_s(n)$ can be obtained from the Zimin word $\mathcal{Z}_n$ by mapping X_i's to a_i's. Despite the similarities, we prefer to introduce this new pattern, as the advantage of $\mathcal{C}(n)$ is that it allows for the succinct expression of the most general offending suffix possible.

Given a list L, let $h : \Delta^* \to \Sigma_L^*$, be a morphism. We say that h *respects* a list $L = L_1, L_2, \ldots, L_n$, if h yields a string over L. So, for example, an h that maps each $\boldsymbol{X}_1, \boldsymbol{X}_2, \boldsymbol{X}_3$ to ε, and also maps $a_1 \mapsto a, a_2 \mapsto b, a_3 \mapsto c$, yields $h(\mathcal{C}(3)) = abacaba$. Such an h respects, for example, a list $L = \{a, e\}, \{a, b\}, \{a, d\}, \{c\}, \{a, e\}, \{b, c, d\}, \{a\}$.

The main result of the paper, a characterization of squares in strings over lists in terms of offending suffixes, follows.

Lemma 26. *Suppose that $\boldsymbol{w} = \boldsymbol{w}_1\boldsymbol{w}_2 \ldots \boldsymbol{w}_{i-1}$ is a square-free string over a list $L = L_1, L_2, \ldots, L_{i-1}$, where $L \in \mathcal{L}_3$. Then, the pivots $L_i = \{a, b, c\}$ force a square on $\boldsymbol{w}$ iff $\boldsymbol{w}$ has a suffix conforming to the offending suffix $\mathcal{C}(3)$.*

Proof. The proof is by contradiction. We assume throughout that our lists are from the class $\mathcal{L}_3$.

($\Leftarrow$) Suppose $\boldsymbol{w} = \boldsymbol{w}_1\boldsymbol{w}_2 \ldots \boldsymbol{w}_{i-1}$ has a suffix conforming to the offending suffix $\mathcal{C}(3)$, where a, b, c are the pivots. Clearly, if we let $L_i = \{a, b, c\}$, then each $\boldsymbol{w}a, \boldsymbol{w}b, \boldsymbol{w}c$ has a square, and hence by definition L_i forces a square on $\boldsymbol{w}$.

($\Rightarrow$) Suppose, on the other hand, that $L_i = \{a, b, c\}$ forces a square on the word $\boldsymbol{w}$ over $L = L_1, L_2, \ldots, L_{i-1}$. We need to show that $\boldsymbol{w}$ must have a suffix that conforms to the pattern $\mathcal{C}(3)$, with the symbols a, b, c as the pivots. Since L_i forces a square, we know that $\boldsymbol{w}a, \boldsymbol{w}b, \boldsymbol{w}c$ has a square for a suffix (as $\boldsymbol{w}$ itself was square-free). Let $\boldsymbol{t}a\boldsymbol{t}a, \boldsymbol{u}b\boldsymbol{u}b, \boldsymbol{v}c\boldsymbol{v}c$ be the squares created by appending a, b and c to $\boldsymbol{w}$, respectively. Here $\boldsymbol{t}, \boldsymbol{u}, \boldsymbol{v}$ are treated as subwords of $\boldsymbol{w}$.

As all three squares $\boldsymbol{t}a\boldsymbol{t}a, \boldsymbol{u}b\boldsymbol{u}b, \boldsymbol{v}c\boldsymbol{v}c$ are suffixes of the string $\boldsymbol{w}$, it follows that $\boldsymbol{t}, \boldsymbol{u}, \boldsymbol{v}$ must be of different sizes, and so we can order them without loss of generality as follows: $|\boldsymbol{t}a\boldsymbol{t}| < |\boldsymbol{u}b\boldsymbol{u}| < |\boldsymbol{v}c\boldsymbol{v}|$. It also follows from the fact that all three are suffixes of $\boldsymbol{w}$, the squares from left-to-right are suffixes of each other. Hence, while $\boldsymbol{t}$ may be empty, we know that $\boldsymbol{u}$ and $\boldsymbol{v}$ are not. We now consider different cases of the overlap of $\boldsymbol{t}a\boldsymbol{t}, \boldsymbol{u}b\boldsymbol{u}, \boldsymbol{v}c\boldsymbol{v}$, showing in each case that the resulting string has a suffix conforming to the pattern $\mathcal{C}(3)$. Note that it is enough to consider the interplay of $\boldsymbol{u}b\boldsymbol{u}, \boldsymbol{v}c\boldsymbol{v}$, as then the interplay of $\boldsymbol{t}a\boldsymbol{t}, \boldsymbol{u}b\boldsymbol{u}$ is symmetric and follows by analogy. Also keep in mind that the assumption is that $\boldsymbol{w}$ is square-free; this eliminates some of the possibilities as can be seen below.

1. $\boldsymbol{v} = \boldsymbol{p}\boldsymbol{u}b\boldsymbol{u}$ as shown in Figure 5.1, where $\boldsymbol{p}$ is a proper non-empty prefix of $\boldsymbol{v}$. Since $\boldsymbol{w}$ is square-free, we assume that $\boldsymbol{p}\boldsymbol{u}b\boldsymbol{u}$ has no square, and therefore $\boldsymbol{p} \neq \boldsymbol{u}$ and $\boldsymbol{p} \neq b$. From this, we get $\boldsymbol{v}c\boldsymbol{v} = \boldsymbol{p}\boldsymbol{u}b\boldsymbol{u}c\boldsymbol{p}\boldsymbol{u}b\boldsymbol{u}$. Therefore, this case is possible.

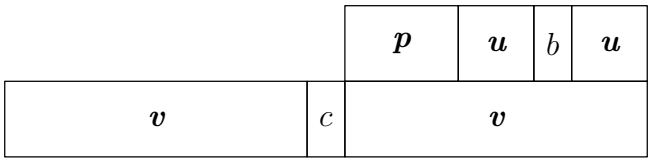

Figure 5.1: $\boldsymbol{v} = \boldsymbol{p}\boldsymbol{u}b\boldsymbol{u}$

2. $\boldsymbol{v} = \boldsymbol{u}b\boldsymbol{u}$ as shown in Figure 5.2. Then, $\boldsymbol{v}c\boldsymbol{v} = \boldsymbol{u}b\boldsymbol{u}c\boldsymbol{u}b\boldsymbol{u}$. This case is also possible.

3. $c\boldsymbol{v} = \boldsymbol{u}b\boldsymbol{u}$ as shown in Figure 5.3, then $\boldsymbol{u}_1 = c$. Let $\boldsymbol{u} = c\boldsymbol{s}$, where $\boldsymbol{s}$ is a proper non-empty suffix of $\boldsymbol{u}$, then $\boldsymbol{v}c\boldsymbol{v} = c\boldsymbol{s}bc\boldsymbol{s}cc\boldsymbol{s}bc\boldsymbol{s}$. The subword '$cc$' indicates a square in $\boldsymbol{w}$. This is a contradiction and therefore this case is not possible.

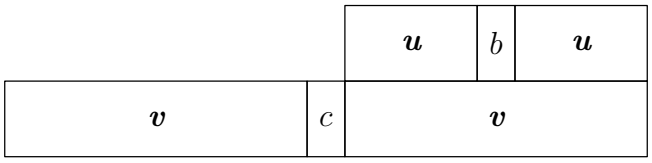

Figure 5.2: $\boldsymbol{v} = \boldsymbol{u}b\boldsymbol{u}$

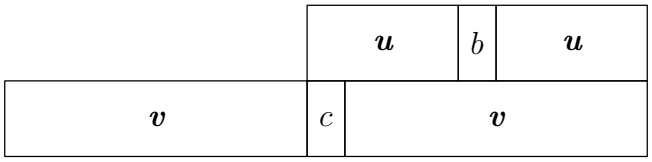

Figure 5.3: $c\boldsymbol{v} = \boldsymbol{u}b\boldsymbol{u}$

4. $\boldsymbol{v}c\boldsymbol{v} = \boldsymbol{q}\boldsymbol{u}b\boldsymbol{u}$ and $|c\boldsymbol{v}| < |\boldsymbol{u}b\boldsymbol{u}|$ as shown in Figure 5.4, where $\boldsymbol{q}$ is a proper prefix of $\boldsymbol{v}c\boldsymbol{v}$. Let $\boldsymbol{u} = \boldsymbol{p}c\boldsymbol{s}$, where $\boldsymbol{p}, \boldsymbol{s}$ are proper prefix and suffix of $\boldsymbol{u}$. Therefore $\boldsymbol{v} = \boldsymbol{s}b\boldsymbol{p}c\boldsymbol{s}$. Since $\boldsymbol{p}$ is also a proper suffix of $\boldsymbol{v}$, one of the following must be true:

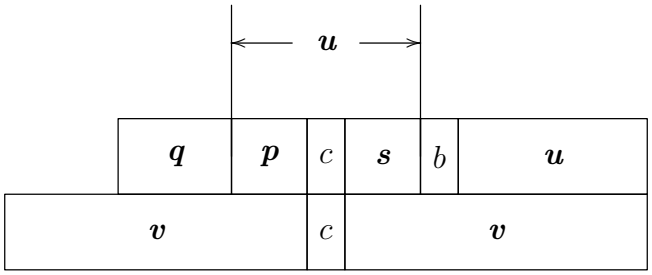

Figure 5.4: $\boldsymbol{v}c\boldsymbol{v} = \boldsymbol{n}\boldsymbol{g}b\boldsymbol{u}$ and $|c\boldsymbol{v}| < |\boldsymbol{u}b\boldsymbol{u}|$

(a) $|\boldsymbol{s}| = |\boldsymbol{p}|$ and so $\boldsymbol{s} = \boldsymbol{p}$. Since $\boldsymbol{s} = \boldsymbol{p}$, $\boldsymbol{v} = \boldsymbol{s}b\boldsymbol{s}c\boldsymbol{s}$ and $\boldsymbol{v}c\boldsymbol{v} = \boldsymbol{s}b\boldsymbol{s}c\boldsymbol{s}c\boldsymbol{s}b\boldsymbol{s}c\boldsymbol{s}$. The subword '$\boldsymbol{s}c\boldsymbol{s}c$', indicates a square in $\boldsymbol{w}$. This is a contradiction and therefore this case is not possible.

(b) $|\boldsymbol{s}| > |\boldsymbol{p}|$ and so $\boldsymbol{s} = \boldsymbol{r}\boldsymbol{p}$, where $\boldsymbol{r}$ is a proper non-empty prefix of $\boldsymbol{s}$. Substituting $\boldsymbol{r}\boldsymbol{p}$ for $\boldsymbol{s}$, we have $\boldsymbol{v} = \boldsymbol{s}b\boldsymbol{p}c\boldsymbol{s} = \boldsymbol{r}\boldsymbol{p}b\boldsymbol{p}c\boldsymbol{r}\boldsymbol{p}$ and $\boldsymbol{v}c\boldsymbol{v} = \boldsymbol{r}\boldsymbol{p}b\boldsymbol{p}c\boldsymbol{r}\boldsymbol{p}c\boldsymbol{r}\boldsymbol{p}b\boldsymbol{p}c\boldsymbol{r}\boldsymbol{p}$. The subword '$c\boldsymbol{r}\boldsymbol{p}c\boldsymbol{r}\boldsymbol{p}$' indicates a square in $\boldsymbol{w}$. This is a contradiction and

therefore this case is not possible.

(c) $|\boldsymbol{s}| < |\boldsymbol{p}|$ and so $\boldsymbol{p} = \boldsymbol{rs}$, where $\boldsymbol{r}$ is a proper non-empty prefix of $\boldsymbol{p}$. Substituting $\boldsymbol{rs}$ for $\boldsymbol{p}$, we have $\boldsymbol{v} = \boldsymbol{sbpcs} = \boldsymbol{sbrscs}$ and $\boldsymbol{vcv} = \boldsymbol{sbrscscsbrscs}$. The subword '$\boldsymbol{scsc}$' indicates a square in $\boldsymbol{w}$. This is a contradiction and therefore this case is not possible.

From the above analysis, we can conclude that for L_i to force a square on a square-free string $\boldsymbol{w}$, it must be the case that $\boldsymbol{v} = \boldsymbol{zubu}$, where $\boldsymbol{z}$ is a prefix (possibly empty) of $\boldsymbol{v}$ and $\boldsymbol{z} \neq \boldsymbol{u}$ and $\boldsymbol{z} \neq b$.

Similarly, we get $\boldsymbol{u} = \boldsymbol{ytat}$, where $\boldsymbol{y}$ is a prefix (possibly empty) of $\boldsymbol{u}$ and $\boldsymbol{y} \neq \boldsymbol{t}$ and $\boldsymbol{z} \neq \boldsymbol{y}$. Substituting values of $\boldsymbol{u}$ in $\boldsymbol{v}$, we get $\boldsymbol{v} = \boldsymbol{zytatbytat}$ and $\boldsymbol{vcv} = \boldsymbol{zytatbytatczytatbytat}$. But $\boldsymbol{vcv}$, is a suffix of the square-free string $\boldsymbol{w}$, and it conforms to the offending suffix $\mathcal{C}(3)$ where the elements a, b, c are the pivots.

Therefore, we have shown that if an alphabet L_i forces a square in a square-free string $\boldsymbol{w}$, then $\boldsymbol{w}$ has a suffix conforming to the offending suffix $\mathcal{C}(3)$. □

From (5.3), we get the size of an offending suffix for lists $L \in \mathcal{L}_3$, and it is given by:

$$|\mathcal{C}(3)| = 2|\boldsymbol{X}_3| + 4|\boldsymbol{X}_2| + 8|\boldsymbol{X}_1| + 7 \tag{5.4}$$

and therefore the size of the shortest offending suffix for lists $L \in \mathcal{L}_3$ is seven. From these observations we have the following Lemma.

Corollary 27. *If L is a list in $\mathcal{L}_3$ of length at most 7, then L is admissible.*

Proof. It follows from Lemma 26, and the fact that the shortest possible offending suffix is of length, $|\mathcal{C}_s(3)| = 7$. (See equation (5.3).) □

Corollary 28. *If $L \in \mathcal{L}_{\Sigma_n}$, where $|L| \leq 2^n - 1$, then L is admissible.*

Proof. The length of the shortest offending suffix $\mathcal{C}_s(n)$ is equal to $2^n - 1$ (See equation (5.2) and the discussion following it). Although Lemma 26 relates to lists $L \in \mathcal{L}_3$, this result can easily be extended to $L \in \mathcal{L}_{\Sigma_n}$. From this and the length of the shortest offending suffix $\mathcal{C}_s(n)$, we can conclude that a square-free string confirming to the pattern $\mathcal{C}_s(n)$ and of length $2^n - 1$ can be generated over lists $L \in \mathcal{L}_{\Sigma_n}$. □

From Lemma 26, we know that an alphabet in a list $L \in \mathcal{L}_3$ can force a square on a square-free string $\boldsymbol{w}$ iff $\boldsymbol{w}$ has a suffix $\boldsymbol{s}$ conforming to the offending suffix $\mathcal{C}(3)$. The question is whether $\boldsymbol{s}$ is unique, that is, does the square-free string $\boldsymbol{w}$ contain more that one suffix that conforms to the offending suffix pattern? In Lemma 29, we show that any square-free string $\boldsymbol{w}$ over $L \in \mathcal{L}_3$ has only one suffix $\boldsymbol{s}$ conforming to the offending suffix (w.r.t fixed pivots) if any, that is $\boldsymbol{s}$ is unique.

Lemma 29. *Suppose $\boldsymbol{w}$ is a square-free string over $L = L_1, L_2, \ldots, L_{n-1}$, and $L \in \mathcal{L}_3$. If $\boldsymbol{w}$ has suffixes $\boldsymbol{s}$,$\boldsymbol{s}'$ conforming to $\mathcal{C}(3)$ with pivots L_n (where $|L_n| = 3$), then $\boldsymbol{s} = \boldsymbol{s}'$.*

Proof. The proof is by contradiction. Suppose that the square-free string $\boldsymbol{w}$ over $L \in \mathcal{L}_3$ has two distinct suffixes $\boldsymbol{s}$ and $\boldsymbol{s}'$ conforming to the offending suffix $\mathcal{C}(3)$ with pivots $L_n = \{a, b, c\}$. That is $\exists h, h(\mathcal{C}(3)) = \boldsymbol{s}$ and $\exists h', h'(\mathcal{C}(3)) = \boldsymbol{s}'$, and $\boldsymbol{s} \neq \boldsymbol{s}'$, and both have pivots in $\{a, b, c\}$. Without loss of generality, we assume that $|\boldsymbol{s}| < |\boldsymbol{s}'|$, and since they are suffixes of $\boldsymbol{w}$, $\boldsymbol{s}$ is a suffix of $\boldsymbol{s}'$. We now examine all possible cases of overlap. Note that $\boldsymbol{s}' = h'(\mathcal{C}(3)) = h'(\boldsymbol{X}_2\mathcal{C}(2)a_3\boldsymbol{X}_2\mathcal{C}(2))$ for some morphism h'. To examine the cases of overlap, let $\boldsymbol{v} = h'(\boldsymbol{X}_2\mathcal{C}(2))$, then $\boldsymbol{s}' = \boldsymbol{v}h'(a_3)\boldsymbol{v}$, where $h'(a_3)$ represents the middle symbol of $\boldsymbol{s}'$. Similarly, the middle symbol of $\boldsymbol{s}$ is represented by $h(a_3)$ for some morphism h. We intentionally use $h'(a_3)$ in $\boldsymbol{s}'$ (and $h(a_3)$ in $\boldsymbol{s}$) as

we want to cover all the six different ways in which the variables a_1, a_2, a_3 are mapped to pivots a, b, c.

1. If $|\boldsymbol{s}| \leq \lfloor |\boldsymbol{s}'|/2 \rfloor$, then $\boldsymbol{v} = \boldsymbol{ps}$ (see Figure 5.5), where $\boldsymbol{p}$ is a prefix of $\boldsymbol{v}$, and $\boldsymbol{ps}h'(a_3)$ is a prefix of $\boldsymbol{s}'$. Observe that, when $|\boldsymbol{s}| = \lfloor |\boldsymbol{s}'|/2 \rfloor$, $\boldsymbol{p} = \varepsilon$. Since $\boldsymbol{s}$ is an offending suffix, we know that $\boldsymbol{s}h'(a_3)$ has a square and hence $\boldsymbol{s}'$ has a square and it follows that $\boldsymbol{w}$ has a square — contradiction.

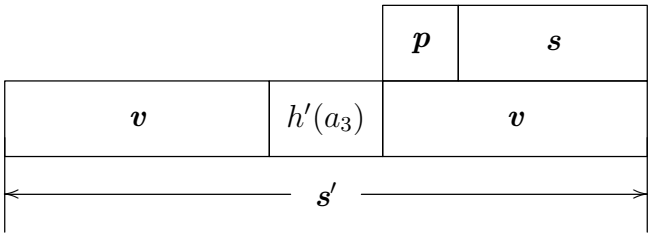

Figure 5.5: $\boldsymbol{v} = \boldsymbol{ps}$

2. If $|\boldsymbol{s}| = \lfloor |\boldsymbol{s}'|/2 \rfloor + 1$, then $\boldsymbol{s} = h'(a_3)\boldsymbol{u}h(a_3)h'(a_3)\boldsymbol{u}$ (see Figure 5.6), where $\boldsymbol{u}$ is a non-empty subword of $\boldsymbol{s}$, and $\boldsymbol{v} = \boldsymbol{u}h(a_3)h'(a_3)\boldsymbol{u}$. If the morphisms h and h' map a_3 to the same element in $\{a, b, c\}$, that is $h'(a_3) = h(a_3)$, then $\boldsymbol{s}$ has a square '$h(a_3)h(a_3)$' and therefore $\boldsymbol{w}$ has a square — contradiction. When $h'(a_3) \neq h(a_3)$, without loss of generality, we assume $h'(a_3) = c$ and $h(a_3) = a$, then $\boldsymbol{v} = \boldsymbol{u}ac\boldsymbol{u}$ and $\boldsymbol{s}' = \boldsymbol{v}c\boldsymbol{v} = \boldsymbol{u}ac\boldsymbol{u}c\boldsymbol{u}ac\boldsymbol{u}$ has a square '$c\boldsymbol{u}c\boldsymbol{u}$' and it follows that $\boldsymbol{w}$ has a square — contradiction.

3. If $|\boldsymbol{s}| > \lfloor |\boldsymbol{s}'|/2 \rfloor + 1$, then $\boldsymbol{s} = \boldsymbol{p}h'(a_3)\boldsymbol{u}h(a_3)\boldsymbol{p}h'(a_3)\boldsymbol{u}$, where $\boldsymbol{p}$ is a non-empty prefix of $\boldsymbol{s}$ and $\boldsymbol{u}$ is a subword (possibly empty) of $\boldsymbol{s}$. Also, $\boldsymbol{v} = \boldsymbol{u}h(a_3)\boldsymbol{p}h'(a_3)\boldsymbol{u}$ and $\boldsymbol{s}' = \boldsymbol{v}h'(a_3)\boldsymbol{v} = \boldsymbol{u}h(a_3)\boldsymbol{p}h'(a_3)\boldsymbol{u}h'(a_3)\boldsymbol{u}h(a_3)\boldsymbol{p}h'(a_3)\boldsymbol{u}$. We can see that $\boldsymbol{s}'$ has a square '$h'(a_3)\boldsymbol{u}h'(a_3)\boldsymbol{u}$', and it follows that $\boldsymbol{w}$ has a square — contradiction.

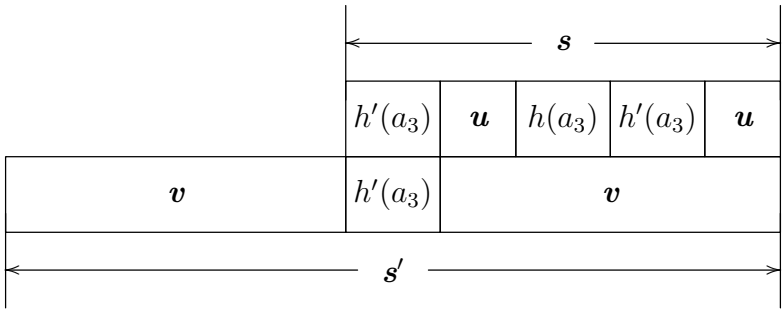

Figure 5.6: $\boldsymbol{v} = \boldsymbol{u}h(a_3)h'(a_3)\boldsymbol{u}$

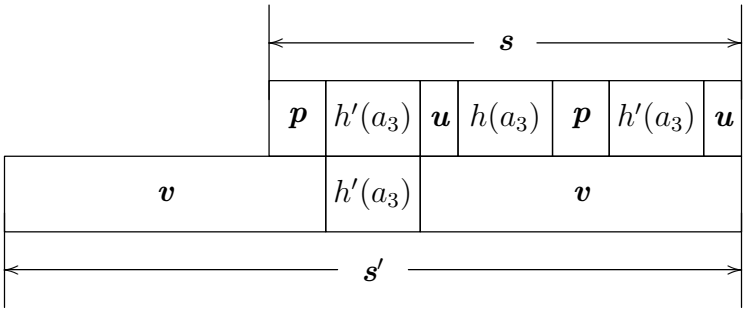

Figure 5.7: $\boldsymbol{v} = \boldsymbol{u}h(a_3)\boldsymbol{p}h'(a_3)\boldsymbol{u}$

This ends the proof. □

5.1.2 Characterization of Square-free Strings

In this section, we give a characterization of square-free strings using borders (see Section 1.4), shown in Lemma 30. There is a lot of literature related to borders (see [Smyth, 2003]), and it seems a plausible future direction that we could build on to design an algorithm for constructing, and therefore proving the existence of square-free strings over lists $L \in \mathcal{L}_3$.

Lemma 30. *A string $\boldsymbol{w}$ is square-free if and only if for every subword $\boldsymbol{s}$ of $\boldsymbol{w}$, if β is a border of $\boldsymbol{s}$ then, $|\beta| < \lceil |\boldsymbol{s}|/2 \rceil$.*

Proof. ($\Rightarrow$) Suppose that $\boldsymbol{s}$ is a subword of $\boldsymbol{w}$ and it has a border β such that $|\beta| \geq$

$\lceil |\boldsymbol{s}|/2 \rceil$. From Figure 5.8 we can see that β must have a prefix $\boldsymbol{p}$ which yields a square $\boldsymbol{pp}$ in $\boldsymbol{s}$ and hence in $\boldsymbol{w}$, and so $\boldsymbol{w}$ is not square-free — contradiction.

($\Leftarrow$) Suppose $\boldsymbol{w}$ has a square $\boldsymbol{s} = \boldsymbol{uu}$. But $\boldsymbol{s}$ is a subword of $\boldsymbol{w}$ and it has a border $\beta = \boldsymbol{u}$ where $|\beta| \geq \lceil |\boldsymbol{s}|/2 \rceil$ — contradiction. □

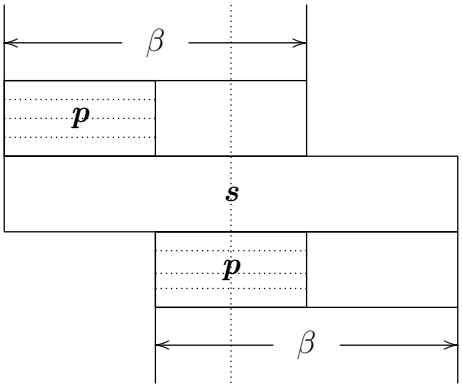

Figure 5.8: $\Rightarrow$ direction of the proof for Lemma 30

This characterization can be used to construct a square-free string $\boldsymbol{w}$, over an alphabet set Σ, where $|\Sigma|$ is sufficiently large, and is as follows: select any symbol from Σ as the first symbol of $\boldsymbol{w}$, then in the i-th step pick the $\boldsymbol{w}_i$-th symbol from Σ such that, the resulting string $\boldsymbol{w}_1\boldsymbol{w}_2 \ldots \boldsymbol{w}_i$ satisfies the border condition in Lemma 30. From the characterization, we know that any string that satisfies the border condition is square-free, therefore $\boldsymbol{w}[1..i]$ is square-free. Also, the sufficiently large alphabet size ensures the existence of an element in Σ, satisfying the border condition.

There are other characterizations for square-free strings with much better running times, for example [Crochemore, 1986] and [Main and Lorentz, 1985] give linear time algorithms to test if a given string is square-free. However, we study this characterization as we intend to use borders and border arrays in conjunction with the offending suffix in investigating the original problem.

5.1.3 0-1 Matrix Representation

In this section, we represent our problem as a 0-1 matrix (matrix consisting of only 0's and 1's). The attraction of this setting is that it may potentially allow us to use the machinery of combinatorial matrix theory to show that $\mathcal{L}_3$ is admissible.

Instead of considering alphabets, we consider sets of natural numbers, i.e., each $L_i \subseteq \mathbb{N}$, and $L = L_1, L_2, \ldots, L_n$, and $\mathcal{L}$ is a class of lists as before. As a consequence, every string over L is a sequence of natural numbers. Note that any $L = L_1, L_2, \ldots, L_n$ can be *normalized* to be $\hat{L}$, where each L_i is replaced with $\hat{L}_i \subseteq [3n]$. This can be accomplished by mapping all integers in $\cup L$, at most $3n$ many of them, in an order preserving way, to $[3n]$. Clearly, L is admissible iff $\hat{L}$ is admissible, and given a list L, it can be normalized in polynomial time.

The integer restatement suggests an approach based on 0-1 matrices. Given a normalized list $\hat{L} = \hat{L}_1, \hat{L}_2, \ldots, \hat{L}_n$, we define the 0-1 $n \times 3n$ matrix A_L as follows:

$$A_L = [A_L(i,j)], \text{where } A_L(i,j) = \begin{cases} 1, & \text{if } j \in \hat{L}_i \\ 0, & \text{if } j \notin \hat{L}_i \end{cases}$$

Observe that, for any normalized list $\hat{L}$, where $L \in \mathcal{L}_3$, each row of the matrix A_L consists of exactly three one's, and the number of one's in each column represents the total number of times the element j appears in the list $\hat{L}$, that is, how many alphabets contain j.

It is easy to see that $\hat{L}$ is admissible iff there is a selection S that picks a single 1 in each row in such a way that there are no i consecutive rows equal to the next i consecutive rows. More precisely, $\hat{L}$ is admissible iff there does not exist i, j, such that

$1 \leq i \leq j \leq \lfloor \frac{n}{2} \rfloor$, and such that the sub matrix of A_L consisting of rows i through j is equal to the sub matrix of A_L consisting of rows $j+1$ through $2(j+1)-i$.

Suppose that $\Sigma_{\hat{L}}$ is re-ordered bijectively by π, where π is a permutation, that is, a bijection $\pi : [n] \longrightarrow [n]$. We define $\pi(\hat{L})$ as follows: for all alphabet in $\hat{L}$, if $\hat{L}_i = \{j, k, \ell\}$, then $\pi(\hat{L}_i) = \{\pi(j), \pi(k), \pi(\ell)\}$. Therefore,

$$\pi(\hat{L}) = \{\pi(\hat{L}_1), \pi(\hat{L}_2), \ldots, \pi(\hat{L}_n)\}.$$

Claim 31. *$\hat{L}$ is admissible iff $\pi(\hat{L})$ is admissible.*

Proof. ($\Rightarrow$) Suppose $\hat{L}$ is admissible. Then there is a square-free string $\boldsymbol{w} = \boldsymbol{w}_1 \boldsymbol{w}_2 \ldots \boldsymbol{w}_n$ of length n over $\hat{L}$. Let

$$\boldsymbol{w}' = \pi(\boldsymbol{w}_1)\pi(\boldsymbol{w}_2)\ldots\pi(\boldsymbol{w}_n),$$

then $\boldsymbol{w}'$ is a string over $\pi(\hat{L})$, and it is also non-repetitive. If it were the case that $\boldsymbol{vv}$ was a subword of $\boldsymbol{w}'$, then $\boldsymbol{vv}$ under π^{-1} would be a square in $\boldsymbol{w}$, which is a contradiction.

($\Leftarrow$) Suppose $\pi(\hat{L})$ is admissible. Then there is a square-free string $\boldsymbol{w} = \boldsymbol{w}_1 \boldsymbol{w}_2 \ldots \boldsymbol{w}_n$ of length n over $\pi(\hat{L})$. Let

$$\boldsymbol{w}' = \pi^{-1}(\boldsymbol{w}_1)\pi^{-1}(\boldsymbol{w}_2)\ldots\pi^{-1}(\boldsymbol{w}_n),$$

then $\boldsymbol{w}'$ is a string over $\hat{L}$, and it is also non-repetitive. If it were the case that $\boldsymbol{vv}$ was a subword of $\boldsymbol{w}'$, then $\boldsymbol{vv}$ under π would be a square in $\boldsymbol{w}$, which is a contradiction. □

Note that a bijective re-ordering of $\Sigma_{\hat{L}}$ is represented by a permutation of the columns of A_L. Thus, permuting the columns of A_L does not really change the

problem; the same is not true of permuting the rows, which actually re-orders the list L, changing the constraints, and therefore changing the problem.

5.1.4 Proof Complexity

By restating the generalized Thue problem in the language of 0-1 matrices, as we did in Section 5.1.3, we can more easily formalize the relevant concepts in the language of first order logic, and use its machinery to attack the problem.

We adopt the logical theory $\mathbf{V}^0$ as presented in [Cook and Nguyen, 2010], whose language is $\mathcal{L}_A^2 = [0, 1, +, \cdot, ||; =_1, =_2, \leq, \in]$ (see [Cook and Nguyen, 2010, Definition IV.2.2, pg. 76]). Without going into all the details, this language allows the indexing of a 0-1 string $\boldsymbol{x}$; on the other hand, a 0-1 matrix A_L can be represented as a string $\boldsymbol{x}_L$ with the definition: $\boldsymbol{x}_L(3n(i-1)+j) = A_L(i,j)$. Hence, $\mathcal{L}_A^2$ is eminently suitable for expressing properties of strings.

Define the following auxiliary predicates:

- Let $\text{Three}(\boldsymbol{x}_L)$ be a predicate which states that the matrix A_L corresponding to $\boldsymbol{x}_L$ has exactly three 1s per row.
- Let $\text{Sel}(Y_L, \boldsymbol{x}_L)$ be a predicate which states that Y_L is a selection of X_L, in the sense that Y_L corresponds to the 0-1 matrix which selects a single 1 in each row of A_L.
- Let $\text{SF}(Y_L)$ be a predicate which states that Y_L is square-free.

Lemma 32. *All three predicates* $\text{Three}, \text{SF}, \text{Sel}$ *are* Σ_0^B.

Our conjecture, lists in $\mathcal{L}_3$ being admissible, can be stated as a Σ_1^B formula over

$\mathcal{L}_A^2$ as follows:

$$\alpha(\boldsymbol{x}_L) := \exists Y_L = n(\text{Three}(\boldsymbol{x}_L) \wedge \text{Sel}(Y_L, \boldsymbol{x}_L) \wedge \text{SF}(Y_L)).$$

Suppose we can prove that $\mathbf{V}^0 \vdash \alpha(\boldsymbol{x}_L)$; then, we would be able to conclude that given any L, we can compute a square-free string over L in $\mathbf{AC}^0$. Likewise, if $\mathbf{V}^1 \vdash \alpha(\boldsymbol{x}_L)$, then we would be able to conclude that the non-repetitive string can be computed in polynomial time.

5.1.5 Open Problems

We conclude this section with the following list of open problems:

1. Is the class of lists, $\mathcal{L}_3$, admissible?

2. How can we de-randomize [Grytczuk et al., 2013, Algorithm 1] in polynomial time? The naïve way to de-randomize it is to employ an exhaustive search algorithm: given an L in $\mathcal{L}_4$, examine every $\boldsymbol{w} \in L^+$ in lexicographic order until a square-free string is found, which by [Grytczuk et al., 2013, Theorem 1] must happen. In that sense, the correctness of the probabilistic algorithm implies the correctness of the deterministic exhaustive search algorithm. However, such an exhaustive search algorithm takes $4^{|L|}$ steps in the worst case. Is it possible to de-randomize it, to a deterministic polytime algorithm?

3. What is the relationship between admissible $\mathcal{L}$ in the original sense, and those $\mathcal{L}$ for which the player has a winning strategy in the online game sense. Clearly, if there exists a winning strategy for $\mathcal{L}$, then $\mathcal{L}$ is admissible; what about the converse?

5.2 String Shuffle: Circuits and Graphs

In the area of studying properties of string shuffle and determining tighter complexity lower bound for the shuffle problem, we propose a formulation of the shuffle problem as an instance of a 0-1 matrix problem, and hope to use the tools of combinatorial matrix theory to determine a tighter circuit complexity lower bound, if one exists. In [Soltys, 2013] and [Mhaskar and Soltys, 2015c] it has been shown that shuffle can be decided in $\mathbf{AC}^1$, but, can we given a tighter upper bound and show that Shuffle $\in \mathbf{NC}^1$?

Given strings $\boldsymbol{u}, \boldsymbol{v}$ and $\boldsymbol{w}$, where $|\boldsymbol{u}| = |\boldsymbol{v}| = \frac{|\boldsymbol{w}|}{2} = n$. If a symbol in string $\boldsymbol{u}$ (or $\boldsymbol{v}$) is equal to a symbol in string $\boldsymbol{w}$ we say that the symbols *match* and call it a *matching*. If the j-th symbol in $\boldsymbol{u}$ (or $\boldsymbol{v}$) matches any of the symbols with indices in the set $\{j, j+1, \ldots, j+n\}$ in $\boldsymbol{w}$ then we call this a *valid matching* with respect to $\boldsymbol{u}_j$ (or $\boldsymbol{v}_j$). The set of indices $\{j, j+1, \ldots, j+n\}$ is called a *valid set* with respect to symbol $\boldsymbol{u}_j$ (or $\boldsymbol{v}_j$). For example, if the first symbol of $\boldsymbol{u}$, that is, $\boldsymbol{u}_1$ matches any of the symbols $\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_{n+1}$ then it is a valid matching with respect to $\boldsymbol{u}_1$. The intuition behind this is that when $\boldsymbol{u}$ and $\boldsymbol{v}$ are shuffled to get $\boldsymbol{w}$, it is pointless to check a matching of $\boldsymbol{u}_1$ with $\boldsymbol{w}_{n+2}$-th symbol of $\boldsymbol{w}$ or any symbol further, as less than $n-1$ positions exist in $\boldsymbol{w}$ to accommodate the $n-1$ symbols of $\boldsymbol{u}$.

5.2.1 0-1 Matrix Formulation for Shuffle

In this section we show a 0-1 matrix formulation of the shuffle problem. This entire matrix can be constructed in $\mathbf{NC}^1$. Since boolean addition, product and symmetric

functions are all in $\mathbf{NC}^1$, if we can show that using polynomial number of these operations in parallel we can permute the columns of the matrix such that the resultant permutated matrix has certain desired properties (stated in Claim 33), then we can show that Shuffle $\in \mathbf{NC}^1$. The advantage of this result is that it would mean an efficient parallel algorithm exists for the shuffle problem.

Given three strings $\boldsymbol{u} = \boldsymbol{u}_1\boldsymbol{u}_2 \ldots \boldsymbol{u}_n$, $\boldsymbol{v} = \boldsymbol{v}_1\boldsymbol{v}_2 \ldots \boldsymbol{v}_n$ and $\boldsymbol{w} = \boldsymbol{w}_1\boldsymbol{w}_2 \ldots \boldsymbol{w}_{2n}$ over a binary alphabet, we formulate the question, is $\boldsymbol{w}$ a shuffle of strings $\boldsymbol{u}$ and $\boldsymbol{v}$ as a 0-1 matrix problem. This matrix A_S consists of $3n+1$ rows and $2n$ columns and is defined as follows:

$A_S = [A_S(i,j)]$, where $[A_S(i,j)] =$

$$\begin{cases}
0, & \text{if } i = 1 \ \wedge \ 1 \leq j \leq n \\
1, & \text{if } i = 1 \ \wedge \ n+1 \leq j \leq 2n \\
1, & \text{if } i > j \ \wedge \ 1 < i \leq n+1 \ \wedge \ 1 \leq j \leq n \\
0, & \text{if } i \leq j \ \wedge \ 1 < i \leq n+1 \ \wedge \ 1 \leq j \leq n \\
1, & \text{if } i > j-n \ \wedge \ 1 < i \leq n+1 \ \wedge \ n+1 \leq j \leq 2n \\
0, & \text{if } i \leq j-n \ \wedge \ 1 < i \leq n+1 \ \wedge \ n+1 \leq j \leq 2n \\
1, & \text{if } \boldsymbol{u}_j = \boldsymbol{w}_{i-n-1} \ \wedge \ n+1 < i \leq 3n+1 \ \wedge \ 1 \leq j \leq n \ \wedge \ j \leq i-n-1 \leq n+j \\
0, & \text{if } \boldsymbol{u}_j \neq \boldsymbol{w}_{i-n-1} \ \wedge \ n+1 < i \leq 3n+1 \ \wedge \ 1 \leq j \leq n \ \wedge \ j \leq i-n-1 \leq n+j \\
0, & \text{if } n+1 < i \leq 3n+1 \ \wedge \ 1 \leq j \leq n \ \wedge \ i-n-1 < j \ \vee \ i-n-1 > n+j \\
1, & \text{if } \boldsymbol{v}_{j-n} = \boldsymbol{w}_{i-n-1} \ \wedge \ n+1 < i \leq 3n+1 \ \wedge \ n+1 \leq j \leq 2n \ \wedge \ j-n \leq i-n-1 \leq j \\
0, & \text{if } \boldsymbol{v}_{j-n} \neq \boldsymbol{w}_{i-n-1} \ \wedge \ n+1 < i \leq 3n+1 \ \wedge \ n+1 \leq j \leq 2n \ \wedge \ j-n \leq i-n-1 \leq j \\
0, & \text{if } n+1 < i \leq 3n+1 \ \wedge \ n+1 \leq j \leq 2n \ \wedge \ i-n-1 < j-n \ \vee \ i-n-1 > j
\end{cases}$$

Let C_j be the j-th column vector of A_S. The first n columns represent information about the n symbols of $\boldsymbol{u}$, and the columns $n+1$ to $2n$ represent information about the n symbols of $\boldsymbol{v}$. In particular, the j-th symbol in $\boldsymbol{u}$ ($\boldsymbol{u}_j$) is represented by the j-th column in A_S, and encodes the information regarding its position in $\boldsymbol{u}$ (that is j), followed by the details about valid matchings with symbols in $\boldsymbol{w}$. Similarly, $\boldsymbol{v}_j$ is represented by the $(n+j)$-th column in A_S. Therefore, $C_1, C_2, \ldots, C_n$ column vectors represent symbols $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots \boldsymbol{u}_n$ and column vectors $C_{n+1}, C_{n+2}, \ldots, C_{2n}$ represent symbols $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n$.

The column vectors representing $\boldsymbol{u}$ have their first element 0 and those representing symbols in $\boldsymbol{v}$ have their first element 1. Hence, the first row consists of n zeros followed by n ones. The rows 2 to $n+1$ keep track of the ordering of symbols in $\boldsymbol{u}$ and $\boldsymbol{v}$. Suppose, a symbol appears in the j-th position in $\boldsymbol{u}$, then the $(j+1)$-th row consists of all ones staring from the first column to the j-th column in A_S. The remaining elements of the row upto the n-th column are zero. Thus, forming a lower unit triangular matrix in the sub matrix consisting of $C_1, C_2 \ldots, C_n$ column vectors and rows two to $n+1$. Similarly, if an element appears in the j-th position in $\boldsymbol{v}$, then the $(j+1)$-th row consists of all ones staring from $(n+1)$-th column to the $(n+j)$-th column in A_S. The remaining elements of the row from $(n+j+1)$-th column to $2n$-th column are zero. Thus, forming a lower unit triangular matrix in the sub matrix consisting of $C_{n+1}, C_{n+2}, \ldots, C_{2n}$ column vectors and rows two to $n+1$. Observe that, the position of the j-th symbol in $\boldsymbol{u}$ can also be identified from the C_j-th column vector with its initial symbol being a 0 (indicating that it is a symbol of $\boldsymbol{u}$), followed by $(j-1)$ zeros and $(n-j-1)$ ones.

The sub matrix consisting of $2n$ rows starting form $(n+2)$-th row to $(3n+1)$-th

row encodes the information about valid matchings of symbols in $\boldsymbol{u}, \boldsymbol{v}$ with $\boldsymbol{w}$. In particular, the valid set for symbol $\boldsymbol{u}_j$ or $\boldsymbol{v}_j$ is $\{j, j+1, \ldots, n+j\}$. The valid set for symbol $\boldsymbol{u}_j$ is represented by cells $A_S[n+j+1, j], A_S[n+j+2, j], \ldots, A_S[2n+j+1, j]$, and for the symbol $\boldsymbol{v}_j$ it is represented by cells $A_S[n+j+1, j+n], A_S[n+j+2, j+n], \ldots, A_S[2n+j+1, j+n]$. If a valid matching exists then the corresponding cell is filled with one otherwise it is filled with zero. For example, if $\boldsymbol{u}_j = \boldsymbol{w}_{j+2}$ then the corresponding cell $A_S[n+j+2, j] = 1$, otherwise it is zero. All the other cells that do not correspond to a valid set in the column j of the sub matrix are filled with zero.

Observe that the first $n+1$ rows are the same for any input $\boldsymbol{u}, \boldsymbol{v}$ and $\boldsymbol{w}$, where $|\boldsymbol{u}| = |\boldsymbol{v}| = \frac{|\boldsymbol{w}|}{2} = n$. Therefore, each cell in these rows can be computed in $\mathbf{NC}^1$. Now consider the sub matrix starting from $(n+2)$-th to $(3n+1)$-th row. The cells in this sub matrix that do not represent the valid set with respect to the symbols in $\boldsymbol{u}$ and $\boldsymbol{v}$ are all zero and computed in $\mathbf{NC}^1$. The cells that do represent a valid set with respect to a symbol in $\boldsymbol{u}$ or $\boldsymbol{v}$ can be computed using constant number of boolean product ($\wedge$) and boolean addition ($\vee$) operations. To be precise, if $\boldsymbol{w}_i = \boldsymbol{u}_j$ is a valid matching, then the value of the cell representing it is computed as: $A_S[i+n+1, j] = (\boldsymbol{w}_i \wedge \boldsymbol{u}_j) \vee (\bar{\boldsymbol{w}}_i \wedge \bar{\boldsymbol{u}}_j)$. Therefore, the entire matrix A_S can be computed in $\mathbf{NC}^1$.

Claim 33. *$\boldsymbol{w}$ is a shuffle of $\boldsymbol{u}$ and $\boldsymbol{v}$ if and only if there exists a permutation of columns of A_S such that, the resulting matrix (A_S^P) has the following properties:*

- *The diagonal elements of the sub matrix consisting of rows $n+2$ to $3n+1$ are all one,*
- *The sub matrix consisting of column vectors $C_1, C_2 \ldots, C_n$ and rows 2 to $n+1$ has a unit lower triangular matrix, and*

- *The sub matrix consisting of column vectors* $C_{n+1}, C_{n+2} \ldots, C_{2n}$ *and rows* 2 *to* $n+1$ *has a unit lower triangular matrix.*

Proof. ($\Rightarrow$) If Shuffle($\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}$) holds, then the permutation of columns corresponding to the symbols of $\boldsymbol{u}$ and $\boldsymbol{v}$ in $\boldsymbol{w}$ results in a permutated matrix A_S^P satisfying the conditions in Claim 33. In particular, the ordering of symbols of $\boldsymbol{u}$ and $\boldsymbol{v}$ is maintained in $\boldsymbol{w}$ thus satisfying the last two conditions. The fact that Shuffle($\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}$) holds, indicates that their is a valid matching for symbols in strings $\boldsymbol{u}$ and $\boldsymbol{v}$ with $\boldsymbol{w}$ resulting in all ones in the diagonal elements of the sub matrix consisting of rows $n+2$ to $3n+1$.

($\Leftarrow$) The two unit lower triangular matrix in the sub matrix consisting of rows 2 to $n+1$ ensure the ordering of symbols of $\boldsymbol{u}$ and $\boldsymbol{v}$ in $\boldsymbol{w}$ is maintained. The diagonal elements of the sub matrix consisting of rows $n+2$ to $3n+1$ being all one ensure that a valid matching for all symbols in $\boldsymbol{u}$ and $\boldsymbol{v}$ with $\boldsymbol{w}$ exists. Therefore $\boldsymbol{w}$ is a shuffle of $\boldsymbol{u}$ and $\boldsymbol{v}$. □

To understand the construction of A_S and Claim 33, we provide the following example. Let $\boldsymbol{u} = 10$, $\boldsymbol{v} = 11$ and $\boldsymbol{w} = 1101$. The 0-1 matrix A_S is given below: Observe that, permuting column vectors C_2 with C_3 gives the resulting permuted

$$A_S = \begin{array}{c} \begin{matrix} C_1 & C_2 & C_3 & C_4 \end{matrix} \\ \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{array}$$

Figure 5.9: 0-1 Matrix Construction (A_S) for example.

$$A_S^P = \begin{array}{c} \begin{matrix} C_1 & C_3 & C_2 & C_4 \end{matrix} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{array}$$

Figure 5.10: Permutated Matrix A_S^P for the example.

matrix A_S^P that satisfies the conditions of Claim 33. In particular,

1. Sub matrix consisting of rows 4 to 7 of A_S^P have all the diagonal elements equal to one.

2. Sub matrix consisting of columns C_1 and C_2 (which represent symbols of $\boldsymbol{u}$) and rows two and three of A_S^P has a lower unit triangular matrix, and the sub matrix consisting of columns C_3 and C_4 (which represent symbols of $\boldsymbol{v}$) and rows two and three of A_S^P has a lower unit triangular matrix.

Therefore by Claim 33, Shuffle($\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}$) holds.

5.2.2 Another polytime algorithm for Shuffle

In this section we propose another polytime algorithm for shuffle. The aim is that the new approach could possibly be used to devise an efficient parallel algorithm for shuffle and further show that shuffle is in $\mathbf{NC}^1$.

A simple algorithm for checking whether $\boldsymbol{w}$ is a shuffle of $\boldsymbol{u}$, $\boldsymbol{v}$ is to recursively check if $\boldsymbol{w}_1\boldsymbol{w}_2\ldots\boldsymbol{w}_{i+j}$ is a shuffle of $\boldsymbol{u}_1\boldsymbol{u}_2\ldots\boldsymbol{u}_i$ and $\boldsymbol{v}_1\boldsymbol{v}_2\ldots\boldsymbol{v}_j$ for values of $1 \leq i \leq n$ and

$1 \leq j \leq n$. This algorithm can be represented as a binary tree and works as follows: starting from the root and at each internal node, compare the left most element of $\boldsymbol{w}$ with the left most element of $\boldsymbol{u}$ ($\boldsymbol{v}$), if the symbols match then discard them from the respective strings and continue till either all the symbols of $\boldsymbol{u}$, $\boldsymbol{v}$ and/or $\boldsymbol{w}$ have been discarded or the symbols do not match. In particular, the left arrows correspond to checking whether the left most symbol of $\boldsymbol{w}$ matches the left most symbol in $\boldsymbol{u}$, and the right arrows correspond to checking whether the left most symbol of $\boldsymbol{w}$ matches the left most symbol in $\boldsymbol{v}$. If this tree contains the leaf ϵ,ϵ/ϵ then $\boldsymbol{w}$ is a shuffle of $\boldsymbol{u}$, $\boldsymbol{v}$. Note that, the tree can contain more than one such leaf. The total number of leaves ϵ,ϵ/ϵ represents the number of ways in which $\boldsymbol{w}$ is a shuffle of $\boldsymbol{u}$, $\boldsymbol{v}$. An example of the algorithm working is shown in Figure 5.11 where, $\boldsymbol{u} = 11$, $\boldsymbol{v} = 10$ and $\boldsymbol{w} = 1101$.

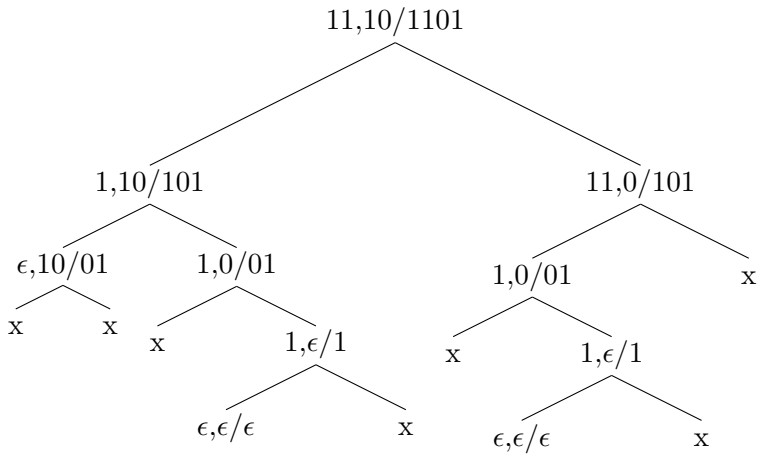

Figure 5.11: Binary tree showing recursive algorithm for shuffle problem.

Since the recursive solution takes exponential time, it is not efficient. A dynamic programming algorithm was given by Mansfield in [Mansfield, 1982], and has a running time of $O(n^2)$. We propose another polytime algorithm for shuffle. We first

define the term matching index and then develop an algorithm (Algorithm 2) to compute a matching index for strings $\boldsymbol{u}, \boldsymbol{v}$ and $\boldsymbol{w}$. If the matching index returned by the algorithm is equal to $|\boldsymbol{w}|$, then $\boldsymbol{w}$ is a shuffle of $\boldsymbol{u}$ and $\boldsymbol{v}$.

A *Matching Index (MI)*, is defined as the length of the longest prefix of $\boldsymbol{w}$ which is a shuffle of a prefix of $\boldsymbol{u}$ and $\boldsymbol{v}$ such that if the prefixes of $\boldsymbol{u}$ and $\boldsymbol{v}$ are of lengths i and j respectively, then the length of the longest prefix of $\boldsymbol{w}$ is $i + j$, that is, the matching index is $i + j$. Therefore, if Shuffle($\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}$) holds, then the *Matching Index* is equal to $2n = |\boldsymbol{w}|$.

Algorithm 2 Matching Index Algorithm

```
1:  procedure MI(u, v, w)
2:      Tag[ ] ← [ ]
3:      if u_1 ≠ w_1 and v_1 ≠ w_1 then
4:          return 0
5:      if u_1 = w_1 then
6:          Tag = Tag ∪ {1}
7:      if v_1 = w_1 then
8:          Tag = Tag ∪ {0}
9:      i ← 2                                   ▷ Looping through symbols of w
10:     while i ≤ 2n do
11:         Temp[ ] ← [ ]
12:         for all j ∈ Tag do
13:             if j < n and u_{j+1} = w_i then
14:                 Temp = Temp ∪ {j + 1}
                        ▷ i − j − 1 = index of symbol in v upto which shuffle holds
15:             if i − j − 1 < n and v_{i−j} = w_i then
16:                 Temp = Temp ∪ {j}
17:         Tag = Temp
18:         if Tag = [ ] then
19:             return i − 1                    ▷ Return the Matching Index till this point
20:         else
21:             i = i + 1                       ▷ Check next symbol of w with symbols in u and v
22:     return i − 1
```

In Algorithm 2, 'Tag' is a set consisting of indices of $\boldsymbol{u}$ upto which the shuffle property holds, that is, if $\boldsymbol{w}_1\boldsymbol{w}_2\ldots\boldsymbol{w}_i$ is a shuffle of $\boldsymbol{u}_1\boldsymbol{u}_2\ldots\boldsymbol{u}_j$ and $\boldsymbol{v}_1\boldsymbol{v}_2\ldots\boldsymbol{v}_{i-j}$ then Tag consists of all possible values of j for which the above statement holds. The outline of the algorithm is as follows: first check whether the first symbol of $\boldsymbol{u}$ ($\boldsymbol{u}_1$) matches with the first symbol of $\boldsymbol{w}$ ($\boldsymbol{w}_1$), if it does then add one to the set Tag. Repeat this with the first symbol of $\boldsymbol{v}$ ($\boldsymbol{v}_1$) and $\boldsymbol{w}_1$, if they match add zero to Tag. If the first symbol of $\boldsymbol{w}$ is not equal to the first symbol of $\boldsymbol{u}$ or $\boldsymbol{v}$ return 0. At each iteration of the while loop we keep checking if the prefix $\boldsymbol{w}_1\boldsymbol{w}_2\ldots\boldsymbol{w}_i$ of $\boldsymbol{w}$ is a shuffle of $\boldsymbol{u}_1\boldsymbol{u}_2\ldots\boldsymbol{u}_{j+1}$ and $\boldsymbol{v}_1\boldsymbol{v}_2\ldots\boldsymbol{v}_{i-j-1}$ (or is a shuffle of $\boldsymbol{u}_1\boldsymbol{u}_2\ldots\boldsymbol{u}_j$ and $\boldsymbol{v}_1\boldsymbol{v}_2\ldots\boldsymbol{v}_{i-j}$), if it is we increment i by one, otherwise return the matching index obtained till that point. If the algorithm exits the while loop normally, then the matching index returned is $2n$ and therefore $\boldsymbol{w}$ is a shuffle of $\boldsymbol{u}$ and $\boldsymbol{v}$.

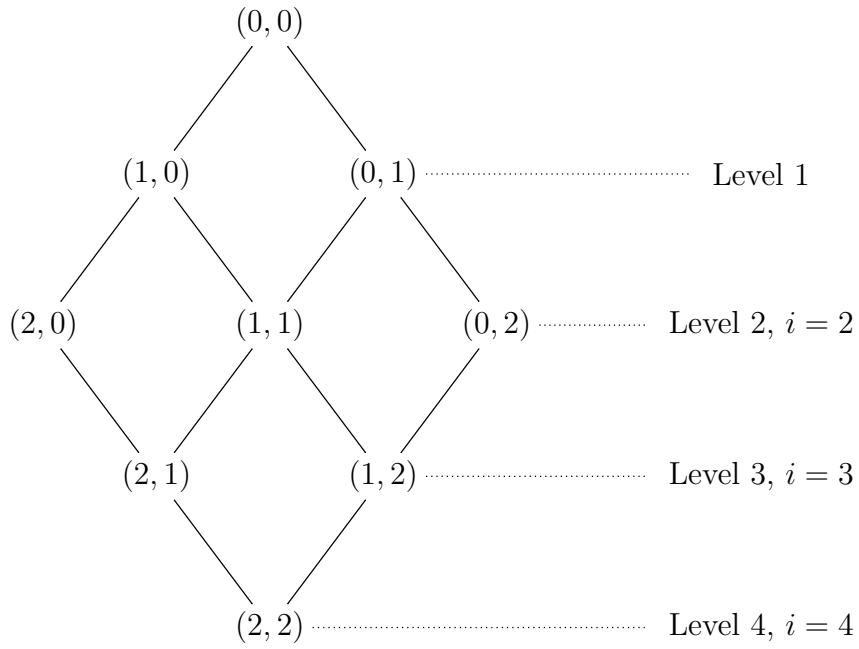

Figure 5.12: Set of x-coordinates represent the possible elements in 'Tag' at the end of each while loop iteration represented by the level.

Figure 5.12 shows all possible ways in which strings $\boldsymbol{u}$ and $\boldsymbol{v}$ can be shuffled to obtain $\boldsymbol{w}$. The levels represent the symbol in $\boldsymbol{w}$ being matched (which corresponds to the value of i at each while loop iteration), and the ordered pairs represent the possible symbol of strings $\boldsymbol{u}$ and $\boldsymbol{v}$ being matched. In particular, the x-coordinate represents the index of the symbol in $\boldsymbol{u}$ being matched and the y-coordinate represents the index of the symbol in $\boldsymbol{v}$ being matched. Therefore the set of all x-coordinates at each level represents the possible values of the elements of set Tag at the end of the corresponding while loop iteration.

In Algorithm 2, the while loop runs for $2n - 1$ times and the set Tag can have at most n (as $|\boldsymbol{u}| = n$) elements in it. Hence, the running time of this algorithm is $O(n^2)$. Therefore, by calling the procedure $MI(\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w})$, we can check whether $\boldsymbol{w}$ is a shuffle of $\boldsymbol{u}, \boldsymbol{v}$ in polynomial time. Although, the running time of the algorithm is not an improvement to the running time known for shuffle ($O(n^2)$), the constants for this algorithm are less and we hope to improve this algorithm further to device an efficient parallel algorithm for shuffle.

5.2.3 Open problems

We conclude this section with a list of open problems in the area.

1. Can shuffle be decided in $\mathbf{NC}^1$?

2. Can we characterize graphs with strings having shuffle properties where more repetitions of symbols are allowed?

3. Regarding Lemma 12, can we test in polynomial time whether a given graph or its complement are isomorphic to the transitive closure of a tree?

5.3 Formal framework for stringology

In Chapter 4, we mentioned the richness of the field of Stringology arises from the fact that a string $\boldsymbol{u}$ is a map $I \longrightarrow \Sigma$, where I can be arbitrarily large, while Σ is small. This produces repetitions and patterns that are the object of study for Stringology. On the other hand, Proof Complexity has studied in depth the varied versions of the Pigeonhole Principle that is responsible for these repetitions. Thus the two may enrich each other.

We have so far only presented the ground work for the formalism for strings. As the first few steps towards the future direction, Lemma 22 can likely be strengthened to saying that evaluating $\mathcal{L}_S$-terms can be done in $\mathbf{AC}^0$ rather than polytime. Also, giving an application of the Witnessing theorem, will provide a deep insight into the nature of strings. One such application can be found in the Lyndon decomposition of a string (see [Smyth, 2003, pg. 29]). Recall that our alphabet is ordered, that is, $\sigma_0 < \sigma_1 < \sigma_2 \ldots$. Hence, we can easily define a lexicographic ordering of strings, by defining a predicate $\boldsymbol{u} <_{\text{lex}} \boldsymbol{v}$. We can define a Lyndon word with a Σ_0^B formula as follows: $\forall i < |\boldsymbol{v}|(\boldsymbol{v} <_{\text{lex}} \lambda k\langle i, e(\boldsymbol{v}, |\boldsymbol{v}| - i + 1 + k)\rangle)$.

Let $\boldsymbol{v}$ be a string; then $\boldsymbol{v} = \boldsymbol{v}_1 \cdot \boldsymbol{v}_2 \cdot \ldots \cdot \boldsymbol{v}_k$ is a Lyndon decomposition if each $\boldsymbol{v}_i$ is a Lyndon word, and $\boldsymbol{v}_k <_{\text{lex}} \boldsymbol{v}_{k-1} <_{\text{lex}} \cdots <_{\text{lex}} \boldsymbol{v}_1$. The existence of a Lyndon decomposition can be proven as in [Smyth, 2003, Theorem 1.4.9], and we assert that the proof itself can be formalized in $\mathcal{S}_1$. We can therefore conclude that the actual decomposition can be computed in polytime.

References

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

[Alon and Spencer, 2008] Alon, N. and Spencer, J. H. (2008). *The probabilistic method,*. Wiley-Interscience Series in Discrete Mathematics and Optimization, John Wiley and Sons Inc.,Hoboken, NJ, third edition.

[Aršon, 1937] Aršon, S. (1937). Proof of the existence of asymmetric infinite sequences (russian). *Mat. Sbornik*, 2:769779.

[Bean et al., 1979] Bean, D. R., Ehrenfeucht, A., and McNulty, G. F. (1979). Avoidable patterns in strings of symbols. *Pacific Journal of Mathematics*, 85(2):261–294.

[Beck, 1991] Beck, J. (1991). An algorithmic approach to the lovász local lemma. *Random Structures and Algorithms*, 2(4):343–365.

[Berstel, 1979] Berstel, J. (1979). Sur les mots sans carré définis par un morphisme. In *Proc. 6th ICALP Symposium*, volume 71 of *Lecture Notes in Computer Science*, pages 16–25. Springer Berlin Heidelberg.

[Berstel, 1995] Berstel, J. (1995). Axel Thue's papers on repetitions in words: a translation. Technical report, Université du Québec a Montréal.

[Berstel et al., 2008] Berstel, J., Lauve, A., Reutenauer, C., and Saliola, F. V. (2008). *Combinatorics on Words: Christoffel Words and Repetitions in Words*. American Mathematical Society.

[Berstel and Perrin, 2007] Berstel, J. and Perrin, D. (2007). The origins of combinatorics of words. *Electronic Journal of Combinatorics*, 28:996–1022.

[Burkhard et al., 1998] Burkhard, R. E., Çela, E., Rote, G., and Woeginger, G. J. (1998). The quadratic assignment problem with a monotone anti-Monge and a symmetric Toeplitz matrix: Easy and hard cases. *Mathematical Programming*, 82:125–158.

[Buss, 1998] Buss, S. R. (1998). An introduction to proof theory. In Buss, S. R., editor, *Handbook of Proof Theory*, pages 1–78. North Holland.

[Buss and Soltys, 2013] Buss, S. R. and Soltys, M. (2013). Unshuffling a square is NP-hard. *Journal of Computer and System Sciences*, 80(4):766–776.

[Buss and Yianilos, 1998] Buss, S. R. and Yianilos, P. N. (1998). Linear and $O(n \log n)$ time minimum-cost matching algorithms for quasi-convex tours. *SIAM J. Comput.*, 27(1):170–201.

[Cook, 1971] Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA. ACM.

[Cook and Nguyen, 2010] Cook, S. A. and Nguyen, P. (2010). *Logical Foundations of Proof Complexity*. Cambridge Univeristy Press.

[Cooper and Rorabaugh, 2014] Cooper, J. and Rorabaugh, D. (2014). Bounds on zimin word avoidance. *Electronic Journal of Combinatorics*, 21(1).

[Crochemore, 1982] Crochemore, M. (1982). Sharp characterizations of squarefree morphisms. *Theoretical Computer Science*, 18:221–226.

[Crochemore, 1986] Crochemore, M. (1986). Transducers and repetitions. *Theoretical Computer Science*, 12:63–86.

[Crochemore and Rytter, 1994] Crochemore, M. and Rytter, W. (1994). *Text Algorithms*.

[Currie, 1991] Currie, J. D. (1991). Which graphs allow infinite non-repetitive walks? *Discrete Mathematics*, 87:249–260.

[Dvořák et al., 2011] Dvořák, Z., Kawarabayashi, K.-I., and Thomas, R. (2011). Three-coloring triangle-free planar graphs in linear time. *ACM Trans. Algorithms*, 7(4):41:1–41:14.

[Erdős and Lovász, 1975] Erdős, P. and Lovász, L. (1975). Problems and results on 3-chromatic hypergraphs and some related questions. *Infinite and Finite Sets, Coll. Math. Soc. J. Bolyai*, 11:363–375.

[Furst et al., 1984] Furst, M., Saxe, J. B., and Sipser, M. (1984). Parity, circuits, and the polynomial-time hierarchy. *Math. Systems Theory*, 17:13–27.

[Garey et al., 1976] Garey, M., Johnson, D., and Stockmeyer, L. (1976). Some simplified np-complete graph problems. *Theoretical Computer Science*, 1(3):237 – 267.

[Ginsburg and Spanier, 1965] Ginsburg, S. and Spanier, E. (1965). Mapping of languages by two-tape devices. *Journal of the Association of Computing Machinery*, 12(3):423–434.

[Gischer, 1981] Gischer, J. (1981). Shuffle languages, Petri nets, and context-sensivite grammars. *Communications of the ACM*, 24(9):597–605.

[Grötzsch, 1959] Grötzsch, H. (1959). Ein dreifarbensatz für dreikreisfreie netze auf der kugel. 8:109–120.

[Gruber and Holzer, 2009] Gruber, H. and Holzer, M. (2009). Tight bounds on the descriptional complexity of regular expressions. In *Proc. Intl. Conf. on Developments in Language Theory (DLT)*, pages 276–287. Springer Verlag.

[Grytczuk et al., 2013] Grytczuk, J., Kozik, J., and Micek, P. (2013). A new approach to nonrepetitive sequences. *Random Structures and Algorithms*, 42:214–225.

[Hall, 1987] Hall, P. (1987). On representatives of subsets. In Gessel, I. and Rota, G.-C., editors, *Classic Papers in Combinatorics*, Modern Birkhäuser Classics, pages 58–62. Birkhäuser Boston.

[J. Grytczuk and Micek, 2011] J. Grytczuk, J. K. and Micek, P. (2011). Nonrepetitive games.

[J. Grytczuk and Zhu, 2011] J. Grytczuk, J. P. and Zhu, X. (2011). Nonrepetitive list colorings of paths. *Random Structures Algorithms*, 38:162–173.

[Jantzen, 1981] Jantzen, M. (1981). The power of synchronizing operations on strings. *Theoretical Computer Science*, 14:127–154.

[Jantzen, 1985] Jantzen, M. (1985). Extending regular expressions with iterated shuffle. *Theoretical Computer Science*, 38:223–247.

[Jedrzejowicz, 1999] Jedrzejowicz, J. (1999). Structural properties of shuffle automata. *Grammars*, 2(1):35–51.

[Jedrzejowicz and Szepietowski, 2001] Jedrzejowicz, J. and Szepietowski, A. (2001). Shuffle languages are in P. *Theoretical Computer Science*, 250(1-2):31=53.

[Jedrzejowicz and Szepietowski, 2005] Jedrzejowicz, J. and Szepietowski, A. (2005). On the expressive power of the shuffle operator matched with intersection by regular sets. *Theoretical Informatics and Applications*, 35:379–388.

[Karhumäki, 2004] Karhumäki, J. (2004). Combinatorics on words: A new challenging topic. Technical Report 645, Turku Center for Computer Science.

[Kleinberg and Tardos, 2006] Kleinberg, J. and Tardos, É. (2006). *Algorithm Design*. Pearson/Addison-Wesley.

[Kozik and Micek, 2013] Kozik, J. and Micek, P. (2013). Nonrepetitive choice number of trees.

[Leech, 1957] Leech, J. (1957). A problem on strings of beads. *Mathematical Gazette*, page 277.

[Levin, 1973] Levin, L. (1973). Universal search problems. *Problems of Information Transmission (translated from Russian)*, 9:115–116.

[Lothaire, 1983] Lothaire, M. (1983). *Combinatorics on Words*, volume 17 of *Encyclopedia of Mathematics*. Addison-Wesley.

[Lothaire, 2002] Lothaire, M. (2002). *Algebraic Combinatorics on Words*, volume 90 of *Encyclopedia of Mathematics*. Cambridge University Press.

[Lothaire, 2005] Lothaire, M. (2005). *Applied Combinatorics on Words*, volume 105 of *Encyclopedia of Mathematics*. Cambridge University Press.

[Main and Lorentz, 1985] Main, M. G. and Lorentz, R. J. (1985). Linear time recognition of squarefree strings. In *Combinatorial Algorithms on Words*, volume F12 of NATO ASI Series, pages 271–278. Springer.

[Mansfield, 1982] Mansfield, A. (1982). An algorithm for a merge recognition problem. *Discrete Applied Mathematics*, 4(3):193–197.

[Mansfield, 1983] Mansfield, A. (1983). On the computational complexity of a merge recognition problem. *Discrete Applied Mathematics*, 1(3):119–122.

[Mayer and Stockmeyer, 1994] Mayer, A. J. and Stockmeyer, L. J. (1994). The complexity of word problems — this time with interleaving. *Information and Computation*, 115:293–311.

[Morse and Hedlund, 1944] Morse, M. and Hedlund, G. A. (1944). Unending chess, symbolic dynamics and a problem in semigroups. *Duke Math. J*, 11:1–7.

[Moser and Tardos, 2009] Moser, R. A. and Tardos, G. (2009). A constructive proof of the general lovasz local lemma. *CoRR*, abs/0903.0544.

[N. Alon and Riordan, 2002] N. Alon, J. Grytczuk, M. H. l. and Riordan, O. (2002). Nonrepetitive colorings of graphs. *Random Structures Algorithms*, 21:336–346.

[Ogden et al., 1978] Ogden, W. F., Riddle, W. E., and Rounds, W. C. (1978). Complexity of expressions allowing concurrency. In *Proc. 5th ACM Symposium on Principles of Programming Languages (POPL)*, pages 185–194.

[Papadimitriou, 1994] Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley.

[Pegden, 2009] Pegden, W. (2009). Highly nonrepetitive sequences: winning strategies from the local lemma. *Random Structures Algorithms (to appear)*.

[Rampersad, 2007] Rampersad, N. (2007). *Overlap-free words and generalizations*. PhD thesis, Waterloo University.

[Rampersad and Shallit, 2012] Rampersad, N. and Shallit, J. (2012). Repetitions in words.

[Riddle, 1973] Riddle, W. E. (1973). A method for the description and analysis of complex software systems. *SIGPLAN Notices*, 8(9):133–136.

[Riddle, 1979] Riddle, W. E. (1979). An approach to software system modelling and analysis. *Computer Languages*, 4(1):49–66.

[Rizzi and Vialette, 2013] Rizzi, R. and Vialette, S. (2013). On recognizing words that are squares for the shuffle product. In *Computer Science – Theory and Applications, 8th International Computer Science Symposium in Russia, CSR*, Lecture Notes in Computer Science 7913, pages 235–245.

[Shallit, 2009] Shallit, J. (2009). *A second course in formal languages and automata theory*. Cambridge Univeristy Press.

[Shaw, 1978] Shaw, A. C. (1978). Software descriptions with flow expressions. *IEEE Transactions on Software Engineering*, SE-4(3):242–254.

[Shoudai, 1992] Shoudai, T. (1992). A P-complete language describable with iterated shuffle. *Information Processing Letters*, 41(5):233–238.

[Sipser, 2013] Sipser, M. (2013). *Introduction to the Theory of Computation*. Cengage Learning.

[Smyth, 2003] Smyth, B. (2003). *Computing Patterns in Strings*. Pearson Education.

[Soltys, 1999] Soltys, M. (1999). A model-theoretic proof of the completeness of LK proofs. Technical Report CAS-06-05-MS, McMaster University.

[Soltys, 2009] Soltys, M. (2009). *An introduction to computational complexity*. Jagiellonian University Press.

[Soltys, 2012] Soltys, M. (2012). *An introduction to the analysis of algorithms*. World Scientific, second edition.

[Soltys, 2013] Soltys, M. (2013). Circuit complexity of shuffle. In Lecroq, T. and Mouchard, L., editors, *International Workshop on Combinatorial Algorithms 2013*, volume 8288 of *Lecture Notes in Computer Science*, pages 402—411. Springer.

[Soltys and Cook, 2004] Soltys, M. and Cook, S. (2004). The proof complexity of linear algebra. *Annals of Pure and Applied Logic*, 130(1–3):207–275.

[Stanely, 2015] Stanely, R. P. (2015). Catalan numbers.

[Stanley, 1999] Stanley, R. P. (1999). *Enumerative Combinatorics*, volume 2. Cambridge Univeristy Press.

[Thomas H. Cormen, 2009] Thomas H. Cormen, Charles E.L̃eiserson, R. L. R. C. (2009). *Introduction to Algorithms*. MIT Press, third edition.

[Thue, 1906] Thue, A. (1906). *Über unendliche zeichenreichen*. Skrifter: Matematisk-Naturvidenskapelig Klasse. Dybwad.

[Thue, 1912] Thue, A. (1912). Über die gegenseitige lage gleicher teile gewisser Zeichenreihen. *Kra. Vidensk. Selsk. Skrifter., I. Mat. Nat. Kl.*, 1:1–67.

[Warmuth and Haussler, 1984] Warmuth, M. K. and Haussler, D. (1984). On the complexity of iterated shuffle. *Journal of Computer and System Sciences*, 28(3):345–358.

[Zimin, 1982] Zimin, A. I. (1982). Blocking sets of terms. *Mat. Sbornik*, 119:363–375.

Co-authored Publications

[Mhaskar and Soltys, 2015a] Mhaskar, N. and Soltys, M. (2015a). A formal framework for stringology. In *Prague Stringology Conference (PSC)*, to appear in Lecture Notes in Computer Science. Springer Berlin Heidelberg.

[Mhaskar and Soltys, 2015b] Mhaskar, N. and Soltys, M. (2015b). Non-repetitive string over alphabet list. In *WALCOM: Algorithms and Computation*, volume 8973 of *Lecture Notes in Computer Science*, pages 270–281. Springer Berlin Heidelberg.

[Mhaskar and Soltys, 2015c] Mhaskar, N. and Soltys, M. (2015c). String shuffle: Circuits and graphs. *Journal of Discrete Algorithms*, 31:120–128.

[Mhaskar and Soltys, 2016] Mhaskar, N. and Soltys, M. (2016). Forced repetitions over alphabet lists. In *Prague Stringology Conference (PSC)*, to appear in Lecture Notes in Computer Science. Springer Berlin Heidelberg.

Appendix A

Main Result of [Grytczuk et al., 2013]

In this section, we restate the main result of [Grytczuk et al., 2013] and give a detailed proof of the result.

Theorem 34 ([Grytczuk et al., 2013], Theorem 1)**.** *Given any alphabet list, $L = L_1, L_2, \ldots, L_n$, where each alphabet is of size at least four, there always is a square-free word over it.*

Proof. The proof is by contradiction. Suppose, it is not possible to construct a square-free string over some list L using Algorithm 1. Then, by this assumption, Algorithm 1 will not terminate on the input L, i.e., the number of steps will be infinite. Let us evaluate the scenario on the m-th step of the algorithm on L. Note that, when we refer to the j-th step of the algorithm, we refer to the j-th iteration of the *while* loop. Also, let us assume that $m > n2^{2n}$, where $n = |L|$.

Let us arbitrarily order the symbols of each alphabet in L. Let $\boldsymbol{r}_j$, where $1 \leq$

$j \leq m$, denote the position of the symbol picked from the corresponding alphabet (this is not necessarily the list L_j) at the j-th step. Therefore, $\boldsymbol{r}_1, \boldsymbol{r}_2 \ldots, \boldsymbol{r}_m$ is the sequence representing the symbols picked from the corresponding alphabet in the steps $1, 2, \ldots, m$ respectively and $1 \leq \boldsymbol{r}_j \leq 4$. This sequence is termed as an *evaluation*. For a fixed evaluation Algorithm 1, becomes deterministic.

For a fixed evaluation $(\boldsymbol{r}_1, \boldsymbol{r}_2 \ldots, \boldsymbol{r}_m)$, let $\boldsymbol{d}_1 = 1$ and $\boldsymbol{d}_j$, where $2 \leq j \leq m$, denote the difference between the values of i at the end of the j-th step and $(j-1)$-th step. $\boldsymbol{d} = \boldsymbol{d}_1, \boldsymbol{d}_2, \ldots, \boldsymbol{d}_m$ is referred to as the *difference sequence*. Note that,

1. $\boldsymbol{d}_j \leq 1$, for all $1 \leq j \leq m$. The idea here is that whenever a symbol is appended to the sequence constructed in Algorithm 1, i is incremented by only one, but when a repetition occurs, the second copy of the repeating word is deleted and hence the value of i can possibly be decremented by more than one depending on the size of the repetition (h).

2. $\Sigma_{j=1}^{k} \boldsymbol{d}_j \geq 1$, for all $1 \leq k \leq m$. The summation of all the values of the difference sequence equals the length of the sequence constructed by the algorithm by step m. Since the first symbol by default has no repetition, the length of the sequence constructed by the algorithm is greater than or equal to one.

The pair $(\boldsymbol{d}, \boldsymbol{s})$ is called the *log* and $\boldsymbol{s} = \boldsymbol{s}_1, \boldsymbol{s}_2 \ldots, \boldsymbol{s}_l$ (where l is the length of the sequence $\boldsymbol{s}$) is the sequence of symbols picked at the end of the m-th step.

Claim 35. *Every log corresponds to a unique evaluation.*

Proof. To prove this claim, we consider the three possible values of $\boldsymbol{d}_j$ and for each case uniquely determine the values of $\boldsymbol{r}_j$. We first look at the last symbol of $\boldsymbol{d}$, i.e.,

$\boldsymbol{d}_m$ and determine the value of $\boldsymbol{r}_m$. Below are the scenarios for the different values of $\boldsymbol{d}_m$:

1. $\boldsymbol{d}_m = 1$: This means that the symbol picked from the L_l-th alphabet ($\boldsymbol{s}_l$) is simply appended to the existing square-free sequence. Thus the value of $\boldsymbol{r}_m$ equals the position of the symbol $\boldsymbol{s}_l$ in the alphabet list L_l, and the sequence at the end of the $(m-1)$-th step is $s_1, s_2, \ldots, s_{l-1}$.

2. $\boldsymbol{d}_m < 0$: This means that the symbol picked from the appropriate alphabet created a repetition, and the repeating block of size $h = |\boldsymbol{d}_m| + 1$ was erased. Since the first part of the repeating block still exists in the sequence $\boldsymbol{s}$, which is the sequence $\boldsymbol{s}_{l-h+1}, \ldots, \boldsymbol{s}_l$, we can use it to re-construct the sequence at the end of $(m-1)$-th step by simply appending the repeating block sans its last element to the sequence $\boldsymbol{s}$. Therefore the sequence at the end of the $(m-1)$-th step is $\boldsymbol{s}_1, \boldsymbol{s}_l, \boldsymbol{s}_{l-h+1}, \ldots, \boldsymbol{s}_{l-1}$ and the value of $\boldsymbol{r}_m$ is the position of the symbol $\boldsymbol{s}_l$ picked from the L_{l+h}-th alphabet.

3. $\boldsymbol{d}_m = 0$: This means that the symbol picked from the alphabet L_{l+1} created a repetition and the repeating block of size $h = |\boldsymbol{d}_m| + 1 = 1$ (in this case the symbol picked) was erased. The value of $\boldsymbol{r}_m$ is the position of the symbol $\boldsymbol{s}_l$ in the L_{l+1}-th alphabet and the sequence at the end of the $(m-1)$-th step is $\boldsymbol{s}_1, \ldots, \boldsymbol{s}_l$.

Applying the same reasoning as above, one can iterate other values of the evaluation from the corresponding symbol in d. □

Let T_m be the number of difference sequences $\boldsymbol{d}$, satisfying the following conditions:

1. $\boldsymbol{d}_j \leq 1$, for all $1 \leq j \leq m$

2. $\Sigma_{j=1}^{k} \boldsymbol{d}_j \geq 1$, for all $1 \leq k \leq m$, and additionally

3. $\Sigma_{j=1}^{m} \boldsymbol{d}_j = 1$

Such sequences are in close relation to planar trees and are enumerated with Catalan numbers.

Catalan numbers form a sequence of natural numbers. Using a zero-based numbering, where the n-th Catalan number is given in terms of binomial coefficients by:

$$\mathcal{C}_n = \frac{1}{n+1}\binom{2n}{n} = \frac{2n!}{(n+1)!(n)!}$$

Asymptotically, the Catalan numbers grow as

$$\mathcal{C}_n \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$$

These sequences occur in various counting problems mostly involving recursively-defined objects. For example, the n-th Catalan number, C_n, is the number of rooted binary trees with n internal nodes. For more details regarding this sequence and its vast application see [Stanely, 2015] and [Stanley, 1999].

Therefore, $T_m = \frac{1}{m+1}\binom{2m}{m}$. Note, that every feasible difference sequence in a log has a total sum less than n (as Algorithm 1 never terminates per our assumption). The number of difference sequences satisfying (i), (ii) but with a total sum equal k (fixed $k \geq 1$) is at most T_m. Thus, the number of all feasible difference sequences is at most $n.T_m = n\frac{n}{m+1}\binom{2m}{m}$. Clearly, for every feasible difference sequence $\boldsymbol{d}$ the number of sequences which can occur in log with $\boldsymbol{d}$ is at most 4^n. The total number of evaluations is 4^m, and from Claim 35, the number of logs equals 4^m. Therefore, we

get the following inequality:

$$4^m \leq n\frac{1}{m+1}\binom{2m}{m}4^n \leq n\frac{4^m}{m\sqrt{2m}}4^n$$

Hence $m\sqrt{2m} \leq n2^{2n}$, which contradicts our assumption. This means that Algorithm 1 terminates. Therefore, square-free strings exist over any $L \in \mathcal{L}_4$. □

To understand the concepts of evaluation, difference sequence, etc. of the proof we consider an example, where the list $L \in \mathcal{L}_{\Sigma_4}$ and $|L| = 8$. We order the symbols of each alphabet in list L in ascending order. Applying the same techniques as discussed in the proof of Theorem 34, we compute the evaluation, difference sequence etc. for the first eight steps. These values are summarized in the table below.

Algorithm 1 **Example**				
Step number(j)	**i (end of step)**	$\boldsymbol{d}_j$	$\boldsymbol{s}$ **(end of step)**	$\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_j$
1	2	1	1	1
2	3	1	12	12
3	4	1	123	123
4	5	1	1231	1231
5	6	1	12312	12312
6	4	-2	123	123123
7	5	1	1231	1231231
8	6	1	12312	12312312

Note that in the above example $\Sigma_{j=1}^{8}\boldsymbol{d}_j = 5$, and this equals the length of the sequence $\boldsymbol{s}$ at the end of the eighth step.