University of Auckland
Faculty of Science
Computer Science

Master Thesis

# The Computation of Finite-State Complexity

by

**Tania K. Roblot**

under the supervision of
Prof. Cristian S. Calude and Dr. Michael J. Dinneen

A thesis submitted in fulfillment of the requirements
for the degree of Master of Science in Computer Science,
The University of Auckland, 2011.

# Abstract

In this thesis, we propose a new variant to Algorithmic Information Theory. This new theory is constructed around finite-state complexity, a computable counterpart to Kolmogorov complexity based on finite transducers rather than Turing machines.

In Chapter 1 we provide the necessary background knowledge and notation for the thesis, mostly stemming from computability theory. We then present finite transducers, the fundamental basis of our new complexity measure, and proffer some recent results about our regular enumerations of these machines. In particular, we introduce our encodings for the set of finite transducers and our main results concerning the definition of such encodings along with the corresponding regular enumerations for this set.

In Chapter 2 we propose the new complexity measure, finite-state complexity, and present our main results concerning its theoretical, computational and practical aspects. The greatest appeal to this new complexity measure is its computability, which we prove and begin to exploit thanks to our main algorithm. This algorithm is presented and its implementation results are used to give an empirical grounding and insight to the theory. The finite-state complexity is also interesting in itself on a computational basis which this thesis will show in some depth, exploring concepts such as incompressibility and state-size hierarchy.

In Chapter 3, we present a first attempt at applying the finite-state complexity in a practical setting: the approximative measure of DNA's finite-state complexity. In sight of the practical limitations we currently face with measuring the finite-state complexity of sequences of such great length, we compromise with an approximative measure of the finite-state complexity of DNA sequences using the results of a DNA-focused grammar-based compressor, Iterative Repeat Replacement Minimal Grammar Parsing, and converting the resulting 'smallest' straight-line grammars into transducers. We also discuss ways in which to optimise our conversion algorithms by heuristic means.

The most of the results in this thesis have been communicated in the following papers: [12], [13], [14], [21] and [33].

*To Alain and Josette Roblot*

# Acknowledgements

This research was initiated during the visit of Professor Kai Salomaa to CDMTCS in 2009. Special thanks go to Prof. K. Salomaa for his continuous and hefty contribution to this work. Part of this research was done during an 8 week internship at INRIA in collaboration with Doctor François Coste in 2010, to whom I am grateful for the opportunity to work abroad on this project. I would like to thank both Dr. F. Coste and Dr. Matthias Gallé for their contribution. I would also like to gratefully acknowledge the Department of Computer Science at the University of Auckland, for the Masters scholarship under which this study was carried out and for allowing me to conduct a short internship abroad. I would also like to acknowledge and thank Alastair Abbott and Sonny Datt for their unwavering support and insight throughout the year(s). I also appreciate A. Abbott, S. Datt, Sophie Lindsay and David Macdonald for taking the time to proof-read this thesis and for their comments. Finally, my gratitude goes to my supervisors, Professor Cristian Calude and Doctor Michael Dinneen, for their guidance and support with this project.

# Contents

# List of Figures

# List of Tables

Chapter 1

# Introduction

The research conducted in this thesis stems directly from computational theory. In this chapter we bring forth the required background and notation used throughout this thesis. We complete this chapter with a detailed presentation of the backbone of our work: finite transducers.

## 1.1 Motivation

Algorithmic Information Theory (AIT) [17, 10] uses various measures of descriptional complexity to define and study different classes of 'algorithmically random' finite strings or infinite sequences. This theory, based on the existence of a universal Turing machine (of various types), is very elegant and has produced many important results, as one can see from the latest monographs on the subject [30, 25].

The incomputability of all descriptional complexities was an obstacle towards more 'down-to-earth' applications of AIT (practical compression, for example). One possibility to avoid incomputability is to restrict the resources available to the universal Turing machine and the result of this is resource-bounded descriptional complexity [8]. Various models which have been studied in this area did not produce significant understandings of 'deterministic randomness' (i.e. chaoticity and software-generated randomness).

Another approach which avoids incomputability involves restricting the computational power of the machines used. For example, the size of the smallest context-free grammar, or straight-line program, generating the singleton language $\{x\}$ is a measure of the descriptional complexity of $x$. This model, investigated since the 1970s, has recently received much attention [20, 29, 28, 34] (also because of connections with Lempel-Ziv encodings [28, 34]). By further restricting the computational power, from context-free grammars to finite automata, one obtains automatic complexity [39]. The automatic complexity of a string is defined as the smallest number of states of a DFA (deterministic finite automaton) that accepts $x$ and does not accept any other string of length $|x|$. Note that a DFA that recognises the singleton language $\{x\}$ always needs $|x| + 1$ states, which is the reason the definition considers only strings of length $|x|$. Automaticity [2, 38] is an analogous descriptional complexity measure for languages.

The first connections between finite-state machine computations and randomness have been obtained for infinite sequences. In [1] it was proved that every subsequence selected from a (Borel) normal sequence by a regular language is also normal. Characterisations of normal infinite sequences have been obtained in terms of finite-state gamblers, information lossless finite-state compressors and finite-state dimension and are defined as follows: a) a sequence is normal if and only if there is no finite-state gambler that succeeds on it [7, 35], and b) a sequence is normal if and only if it is incompressible by any information lossless finite-state compressor [44].

Computations with finite transducers are used in [24] for the definition of finite-state dimension of infinite sequences. The NFA-complexity of a string [20] (non-deterministic finite automata) can be defined in terms of finite transducers that are called "NFAs with advice" [20]; the main problem with this approach is that NFAs used for compression can always be assumed to have only one state.

The finite-state complexity of a finite string $x$ was communicated recently in [12, 33]. It is a complexity measure based on an enumeration of finite transducers and the input strings used by transducers which output a string $x$, following the model used in AIT with the goal to develop a computable version of it. The core purpose of this thesis is to reintroduce and deepen this work.

The main obstacle in developing a version of AIT based on finite transducers is the non-existence of a universal finite transducer. We prove this result later in this thesis. To overcome this negative result we prove the Invariance Theorem for finite-state complexity, based on the fact that we prove that the set of finite transducers can be enumerated by a computable (even a regular) set. The finite-state complexity is computable and examples of finite-state complexities of some strings are presented, along with various properties, most of which were communicated in [12, 13, 14, 21].

## 1.2 Preliminaries

In this section we present some of the general terms used throughout this thesis. Any term or concept that is associated to a specific chapter or section will be introduced accordingly. We follow the formalisations of Sipser [40]. For more thorough introductory details on each of the referenced definitions and concepts, we encourage the reader to refer to Sipser's text [40].

### 1.2.1 Notation

Our notation is standard [6, 10]. If $X$ is a finite set then $X^*$ is the set of all strings (words) over $X$ with $\varepsilon$ denoting the empty string. The length of $x \in X^*$ is denoted by $|x|$.

Throughout this thesis we only consider the binary alphabet, i.e. we take $\Sigma = \{0, 1\}$ unless specified otherwise. Any enumeration done on an alphabet will be in *co-lexicographic* order. By co-lexicographic order we mean that the strings are enumerated in lexicographic order, in sets of growing length. Mathematically, we have the relation $x <_L y$ if $|x| < |y|$ or $|x| = |y|$ and $x_i = 0$ for the smallest $i$ such that $x_i \neq y_i$, where $x = x_1x_2x_3\ldots$ and $y = y_1y_2y_3\ldots$. In other words, for $\Sigma = \{0, 1\}$—recall that we are working with binary—the first few strings in the enumeration are: $\varepsilon$, 0, 1, 00, 01, 10, 11, 000, 001, 010, ...

### 1.2.2 Nomenclature

Here we briefly define the necessary concepts from automata and computability theory.

As expressed in [40], a *grammar* is a collection of (substitution) rules. Each one of these rules is composed of a 'right-hand side' (*rhs*) representing the variable accessing the rule, and a 'left-hand side' (*lhs*) representing the outcome of the rule. The outcome is itself composed of a combination of variables and/or terminals. The terminals are the alphabet characters and cannot be found on the *rhs* of any rule, whereas the purpose of the variables is to indicate the use of an existing rule. One variable is designated to be the start variable and a grammar has completed a configuration, or *derivation*, when there are no more variables to substitute.

A *straight-line grammar* (SLG) is a grammar which can only derive a single unique string [15]. Hence for straight-line grammars each variable is associated with exactly one rule; this guarantees that there can only be one outcome associated with each variable, and therefore with the grammar. Following is the formal definition of a SLG as used in this thesis.

**Definition 1.** A SLG $G$ is a 4-tuple $(V, \Sigma, R, S)$, where:

- $V$ is the finite set of *variables* (or non-terminals);
- $\Sigma$ is the finite alphabet (finite set of *terminals*);
- $R : V \to (V \cup \Sigma)^+$ is the set of *rules* of the form $N_i \to \eta_i$, for $i = 0, \ldots, |R|$, where $\eta_i : (V \cup \Sigma)^+$ and $N_i \notin \eta_i$;[1] and

[1]Since each variable is associated with exactly one rule, i.e. for each $r_i \in R$, $lhs(r_i) \leftrightarrow rhs(r_i)$, each rule is sometimes referred to by its associated variable $S$ or $N_i$ instead of $r_i$. However in this thesis, the distinction is kept for the sake of clarity.

- $S \in V$ is the *start variable.*

**Definition 2** ([40])**.** A *deterministic finite automaton* (DFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

- $Q$ is the finite set of *states*;
- $\Sigma$ is the finite *alphabet*;
- $\delta : Q \times \Sigma \to Q$ is the *transition function*;
- $q_0 \in Q$ is the *start state*; and
- $F \subseteq Q$ is the set of *final* (or accepting) states.

A DFA always halts on all inputs and either accepts or rejects. A well known feature of DFAs is that they recognise exactly the regular languages.

A *finite-state transducer* is a DFA which outputs a string rather than simply *accept* or *reject* [40]. The transition (or transduction) function is a combination of the classical transition function with an output function. Each transition, for a given input symbol, outputs a string and changes the current state to a specific target state. In this thesis we focus on a particular type of finite-state transducers which we explicitly define and present in Section 1.3.

**Definition 3** ([40])**.** A *Turing machine* is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where:

- $Q$ is the finite set of states;
- $\Sigma$ is the finite input alphabet, not containing the *blank symbol* (denoted $\sqcup$);
- $\Gamma$ is the finite tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subset \Gamma$;
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\text{L,R}\}$ is the transition function (where L and R are left and right directions for the tape reader pointer);
- $q_0 \in Q$ is the start state;
- $q_{accept} \in Q$ is the (halting) accept state; and
- $q_{reject} \in Q$ is the (halting) reject state, where $q_{reject} \neq q_{accept}$.

In computability theory, the field of AIT introduced a standardised complexity measure which was defined by Andrei Kolmogorov using Turing machines. Chaitin [16, 17, 18] later developed a variation using the *prefix-free universal Turing machine*, denoted $U$. A *prefix-free* Turing machine is a Turing machine whose language is a prefix-free set of strings. A *universal* Turing machine can simulate every Turing machine, given its description. The complexity measure called *descriptive complexity* (also known as Kolmogorov-Chaitin complexity) [4, 30, 40] of a string $x \in \Sigma^*$ is the function $K : \Sigma^* \to \Sigma^*$ and is defined as follows.

**Definition 4.** Given a string $x \in \Sigma^*$,

$$K(x) = \min_{\langle M,p \rangle \in \Sigma^*} \{\langle M, p \rangle \mid M(p) = x\}$$

where $M$ is a Turing machine ($\langle M \rangle$ being its string description) and $p$ its input string (or program).

Its prefix-free variant, which we will refer to as Kolmogorov complexity in this thesis is the function $H : \Sigma^* \to \Sigma^*$ such that:

**Definition 5.** Given a string $x \in \Sigma^*$,

$$H(x) = \min_{p \in \Sigma^*} \{p \mid U(p) = x\}$$

However, because of its reliance on Turing machines and in particular on the universal Turing machine, Kolmogorov complexity (in all its forms) is confronted with the *Halting problem* and is therefore incomputable [17, 40]. The Halting problem is the inability to decide whether a machine will halt on any given input. Alan Turing [42] proved that this is indeed undecidable over Turing machines. In simple terms, the Halting problem occurs in the context of Turing machines because, unlike DFAs—where the sets of accepting and rejecting states are 'complements' of each other (they are disjoint and their union forms exactly the set of states)—the 'accept' and 'reject' states are two single states within $Q$, not subsets of $Q$. Hence on some inputs those states are unreachable (especially the accept state) and the machine does not halt. Since the universal Turing machine simulates all machines, it obviously is also impaired by this problem, causing the incomputability of this complexity.

*Borel normality* [1, 7, 9, 35, 44] is a concept introduced by Émile Borel in 1909 with his study of reals and sequences. He in fact was the first to explicitly inquire into randomness, despite working with an unsatisfactory definition of randomness [9]. Borel normality allows us to determine if a string or sequence is 'normal' in the sense that all substrings occur within it equally often. For finite strings, it only makes sense to consider substrings of reasonably short length. Let $N_i^m : \Sigma^* \to \mathbb{N}$ count the number of non-overlapping occurrences of the $i$th binary string of length $m$ in a given string. We can then proceed to define Borel normality for strings.

**Definition 6** ([9])**.** Let $x \in \Sigma^*$ be a non-empty string and $1 \le m \le |x|$.

1) We call $x$ *Borel $m$-normal* if for every $1 \le i \le 2^m$ we have:

$$\left| \frac{N_i^m(x)}{\lfloor \frac{|x|}{m} \rfloor} - 2^{-m} \right| \le \sqrt{\frac{\log_2 |x|}{|x|}}.$$

2) If for every natural $1 \leq m \leq \log_2 \log_2 |x|$, $x$ is Borel $m$-normal, then we call $x$ *Borel normal.*

We can easily generalise the above definition to infinite sequences (in fact it was first formulated for infinite sequences) by requiring that it holds in the limit for substrings of increasing length. In this thesis however, our interest relies on finite strings.

In the following section, we thoroughly introduce the machine upon which our complexity measure is based on: finite transducers.

## 1.3 Finite Transducers

### 1.3.1 Definitions

As mentioned in Section 1.2 the complexity we are about to define is based on specific finite transducers. These transducers are in fact a subtype of generalised finite transducers. We give the hierarchy of definitions which leads to our specialised type of machine and which we simply call (with a slight abuse of terminology) a transducer.

**Definition 7.** A *generalised finite transducer* [6] is a 6-tuple $T = (X, Y, Q, q_0, Q_F, E)$, where:

- $X$ is the input alphabet,
- $Y$ the output alphabet,
- $Q$ is the finite set of states,
- $q_0 \in Q$ is the start state,
- $Q_F \subseteq Q$ is the set of accepting states and
- $E \subseteq Q \times X^* \times Y^* \times Q$ is the finite set of transitions.

If $e = (q_1, u, v, q_2) \in E$, $q_1, q_2 \in Q$, $u \in X^*$, $v \in Y^*$ is a transition from $q_1$ to $q_2$, we say that the input (respectively, output) label of $e$ is $u$ (respectively, $v$). When the states are understood from the context we say that the transition $e$ is labeled by $u/v$.

A generalised transducer $T$ whose input alphabet is binary is said to be a *(deterministic sequential) transducer* [6] if it has no transitions with input label $\varepsilon$ and if for any $q \in Q$ and $i \in \Sigma$ there exists a unique $q' \in Q$ and $v \in \Sigma^*$ such that $(q, i, v, q')$ is a transition of $T$. The set of transitions of a deterministic sequential transducer is fully represented by the function

$$\Delta : Q \times \Sigma \to Q \times \Sigma^*. \tag{1.1}$$

For a transducer all states are considered to be final. Hence, we can define a *finite transducer* (transducer for short) as:

**Definition 8.** A *transducer* is a triple $(Q, q_{init}, \Delta)$, where $Q$ is the finite set of states, $q_{init}$ is the initial state and $\Delta$ is the transition function as in (1.1). The function[2] $T : \Sigma^* \to \Sigma^*$ computed by the transducer $(Q, q_{init}, \Delta)$ is defined inductively by

$$T(\varepsilon) = \varepsilon, \quad T(xa) = T(x) \cdot \mu(\hat{\delta}(q_{init}, x), a),$$

where $\delta(q, a) = \pi_1(\Delta(q, a))$, $\mu(q, a) = \pi_2(\Delta(q, a))$, $q \in Q, x \in \Sigma^*$ and $a \in \Sigma$.[3] $\pi_1$ and $\pi_2$ are the first two projections on $Q \times \Sigma^*$. Hence, $\delta : Q \times \Sigma \to Q$ is the first 'component' of $\Delta$, meaning the state projection, and $\mu : Q \times \Sigma \to \Sigma^*$ is the second 'component' of $\Delta$, the output projection. Here $\hat{\delta} : Q \times \Sigma^* \to Q$ is defined inductively by $\hat{\delta}(q, \varepsilon) = q$, $\hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$, where, again, $q \in Q$, and $x \in \Sigma^*$.

A *run* of a machine is a sequence of configurations. A *configuration* of a machine is a description of the 'settings' of the machine at a given instance. These are usually terms used for Turing machines [40] but we can adapt them for simpler machines such as transducers. In terms of transducers, those 'settings' are: the current state, the last read input bit and the current outputted string.

### 1.3.2 Regular Enumerations

We use binary strings to encode transducers and prove that the set of all legal encodings of transducers is a computable and in some cases even a regular language. We encode a transducer by listing for each state $q$ and input symbol $a \in \Sigma$ the output and target state corresponding to the pair $(q, a)$, that is, $\Delta(q, a)$. Thus, the encoding of a transducer is a list of (encodings of) states and output strings.

By $\mathrm{bin}\,(i)$ we denote the binary representation of $i \geq 1$. Note that for all $i \geq 1$, $\mathrm{bin}\,(i)$ always begins with 1; $\mathrm{bin}\,(1) = 1, \mathrm{bin}\,(2) = 10, \mathrm{bin}\,(3) = 11, \ldots$; by $\mathrm{string}\,(i)$ we denote the binary string obtained by removing the leading 1 from $\mathrm{bin}\,(i)$, i.e. $\mathrm{bin}\,(i) = 1 \cdot \mathrm{string}\,(i)$. If $\mathrm{Log}(i) = \lfloor \log_2(i) \rfloor$, then $|\mathrm{string}\,(i)\,| = \mathrm{Log}(i), i \geq 1$.

For $v = v_1 \cdots v_m$, $v_i \in \Sigma$, $i = 1, \ldots, m$, we use the following functions producing self-delimiting versions of their inputs (see [10]): $v^\dagger = v_1 0 v_2 0 \cdots v_{m-1} 0 v_m 1$ and $v^\diamond = \overline{(1v)}^\dagger$, where $^-$ is the negation morphism given by $\overline{0} = 1, \overline{1} = 0$. It is seen that $|v^\dagger| = 2|v|$, and $|v^\diamond| = 2|v| + 2$. In Table 1.1 we present the encodings of the first binary strings.

[2]By slight abuse of notation the function computed by a transducer is named after its associated transducer

[3]In the context of strings, we use $\cdot$ to denote concatenation.

Table 1.1: Encodings of the first binary strings and a comparison of their length, using the different encoding functions: $\mathrm{bin}(n)$, $\mathrm{bin}^\dagger(n)$, $\mathrm{string}(n)$ and $\mathrm{string}^\diamond(n)$. These functions are then used to compose our first encoding $S_0$.

| $n$ | $\mathrm{bin}(n)$ | $\mathrm{bin}^\dagger(n)$ | $\mathrm{string}(n)$ | $\mathrm{string}^\diamond(n)$ | $\lvert\mathrm{bin}^\dagger(n)\rvert = \lvert\mathrm{string}^\diamond(n)\rvert$ |
|---|---|---|---|---|---|
| 1 | 1 | 11 | $\varepsilon$ | 00 | 2 |
| 2 | 10 | 1001 | 0 | 0110 | 4 |
| 3 | 11 | 1011 | 1 | 0100 | 4 |
| 4 | 100 | 100001 | 00 | 011110 | 6 |
| 5 | 101 | 100011 | 01 | 011100 | 6 |
| 6 | 110 | 101001 | 10 | 010110 | 6 |
| 7 | 111 | 101011 | 11 | 010100 | 6 |
| 8 | 1000 | 10000001 | 000 | 01111110 | 8 |

Consider a transducer $T$ with the set of states $Q = \{1, \ldots, n\}$. The transition function $\Delta$ of $T$ (as in (1.1)) is encoded by a binary string

$$\sigma = \mathrm{bin}^\ddagger(i_1) \cdot \mathrm{string}^\diamond(i'_1) \cdot \mathrm{bin}^\ddagger(i_2) \cdot \mathrm{string}^\diamond(i'_2) \cdots \mathrm{bin}^\ddagger(i_{2n}) \cdot \mathrm{string}^\diamond(i'_{2n}), \quad (1.2)$$

where $\Delta(j,k) = \left(i_{2j-1+k} \mod n, \mathrm{string}\left(i'_{2j-1+k}\right)\right)$, $i_t, i'_t \geq 1$, $t = 1, \ldots, 2n$, $j = 1, \ldots, n$, and $k \in \{0,1\}$. We denote by $m \mod n$ the smallest positive integer congruent with $m$ modulo $n$.[4] This allows us to guarantee that our notation respects the size of $Q$, meaning that it will never map a state to a non-existent state in $T$ and similarly with each transition's corresponding output. In (1.2), we modify $\mathrm{bin}^\dagger(i_t)$ into $\mathrm{bin}^\ddagger(i_t)$ to add a 'compacting' feature such that: $\mathrm{bin}^\ddagger(i_t) = \varepsilon$ if the corresponding transition of $\Delta$ is a self-loop, i.e. $\delta(j,k) = \pi_1(\Delta(j,k)) = j$; otherwise, $\mathrm{bin}^\ddagger(i_t) = \mathrm{bin}^\dagger(i_t)$.

The transducer $T$ encoded by $\sigma$ is called $T_\sigma^{S_0}$, where $S_0$ is the set of all strings of the form (1.2) where $1 \leq i_j \leq n$ for all $j = 1, \ldots, 2n$.

**Theorem 9.** *The set of all transducers can be enumerated by a regular language. More precisely, we can construct a regular set $S_0$ such that:*

*a) for every $\sigma \in S_0$, $T_\sigma^{S_0}$ is a transducer,*

*b) for every transducer $T$ one can compute a code $\sigma \in S_0$ such that $T = T_\sigma^{S_0}$.*

*Proof.* We consider the languages $X = \{\mathrm{bin}^\dagger(n) : n \geq 1\} = \{11, 1001, 1011, \ldots\}$, $Y = \{\mathrm{string}^\diamond(n) : n \geq 1\} = \{00, 0110, 0100, \ldots\}$ and we define the language

$$S_0 = (((X \cup \{\varepsilon\})Y)^2)^*. \quad (1.3)$$

[4]In (1.2) we use $i_t$ instead of $i_t \mod n$ in order to guarantee that the set of legal encodings of all transducers is regular, cf. Theorem 9.

The languages $X$ and $Y$ are regular, hence $S_0$ is regular since we can write it in the form (1.3).

The claim a) follows from (1.2). For b) we note that in view of the construction it is clear that every string $\sigma \in S_0$ has a unique factorisation of the form $\sigma = x_1 \cdot y_1 \cdots x_{2n} \cdot y_{2n}$, for appropriate strings $x_1, \ldots, x_{2n} \in X \cup \{\varepsilon\}$ and $y_1, \ldots, y_{2n} \in Y$, since both $X$ and $Y$ are prefix-free sets. So, from $\sigma$ we uniquely get the length $n$ and the codes $x_s \cdot y_s$, for $s = 1, 2, \ldots, 2n$. Every $x_s$ can be uniquely written in the form $x_s = \mathrm{bin}^{\dagger}(t_s)$ and every $y_s$ can be uniquely written in the form $y_s = \mathrm{string}^{\diamond}(r_s)$.

Next we compute the unique transition encoded by $x_s \cdot y_s = \mathrm{bin}^{\dagger}(t_s) \cdot \mathrm{string}^{\diamond}(r_s)$ according to (1.2). First assume that $x_s \neq \varepsilon$. There are two possibilities depending on $s$ being odd or even. If $s = 2i+1$, for $0 \leq i \leq n$, then $\Delta(s,0) = (t_s \mod n, \mathrm{string}(r_s))$; if $s = 2i$, for $1 \leq i \leq n$, then $\Delta(s,1) = (t_s \mod n, \mathrm{string}(r_s))$. The decoding process is unique and shows that the transducer obtained from $\sigma$ is $T_{\sigma}^{S_0} = T$. Secondly, if $x_s = \varepsilon$, then $\Delta(s,0) = (s, \mathrm{string}(r_s))$ for an odd $s$, and $\Delta(s,1) = (s, \mathrm{string}(r_s))$ for an even $s$. □

**Fact 10.** *An explicit encoding $\sigma \in S_0$ of the transition function of $T_{\sigma}^{S_0}$ can be computed in quadratic time.*

**Example 11.** The transducer $T$ with the shortest $S_0$ encoding has one state and always produces the empty string. It has transition function $\Delta : \{1\} \times \Sigma \to \{1\} \times \Sigma^*$ defined by $\Delta(1,0) = \Delta(1,1) = (1, \varepsilon)$. The transducer is coded as $\sigma = \mathrm{bin}^{\ddagger}(1) \cdot \varepsilon^{\diamond} \cdot \mathrm{bin}^{\ddagger}(1) \cdot \varepsilon^{\diamond} = 0000$.

The identity transducer $T_{\mathrm{id}}$ is given by $\Delta(1,0) = (1,0)$, $\Delta(1,1) = (1,1)$. Its code is

$$\sigma_{\mathrm{id}} = \mathrm{bin}^{\ddagger}(1) \cdot 0^{\diamond} \cdot \mathrm{bin}^{\ddagger}(1) \cdot 1^{\diamond} = \varepsilon \cdot 0^{\diamond} \cdot \varepsilon \cdot 1^{\diamond} = 01100100.$$

**Example 12.** Table 1.2, below, has simple examples. The first example is the smallest transducer; the last example is the identity transducer, as explicitly given in Example 11.

The encoding used in Theorem 9 is regular but not as compact as it could be, as the pair $(i, \mathrm{string}(j))$ is coded by $\mathrm{bin}^{\dagger}(i) \cdot \mathrm{string}^{\diamond}(j)$, a string of length $2(\mathrm{Log}(i) + \mathrm{Log}(j)) + 4$.

By using the encoding

$$x^{\S} = 0^{|\mathrm{string}(|x|+1)|} \cdot 1 \cdot \mathrm{string}(|x|+1) \cdot x \tag{1.4}$$

we obtain a more compact one. Indeed, instead of basing our encoding on Table 1.1, we use the functions illustrated in Table 1.3, where

$$\mathrm{string}^{\S}(n) = 0^{|\mathrm{string}(|\mathrm{string}(n)|+1)|} \cdot 1 \cdot \mathrm{string}(|\mathrm{string}(n)|+1) \cdot \mathrm{string}(n),$$

Table 1.2: Some simple transducers encoded using the $S_0$ encodings. This table first shows the $\Delta$ definition of the transducer, then it shows its encoding and finally the length of that particular code.

| transducer | code | code length |
|---|---|---|
| $\Delta_1(1,0) = \Delta_1(1,1) = (1,\varepsilon)$ | $\sigma = 0000$ | 4 |
| $\Delta_1(1,0) = (1,\varepsilon), \Delta_1(1,1) = (1,0)$ | $\sigma = 000110$ | 6 |
| $\Delta_2(1,0) = (1,0), \Delta_2(1,1) = (1,\varepsilon)$ | $\sigma = 011000$ | 6 |
| $\Delta_3(1,0) = \Delta_3(1,1) = (1,0)$ | $\sigma = 01100110$ | 8 |
| $\Delta_4(1,0) = \Delta_4(1,1) = (1,1)$ | $\sigma = 01000100$ | 8 |
| $\Delta(1,0) = (1,0), \Delta(1,1) = (1,1)$ | $\sigma = 01100100$ | 8 |

Table 1.3: Encodings of the first binary strings and a comparison of their length, using the new encoding functions: $\text{bin}^{\#}(n)$ and $\text{string}^{\S}(n)$. These functions are then used to compose our second more compact encoding $S_1$.

| $n$ | $\text{bin}(n)$ | $\text{bin}^{\#}(n)$ | $\text{string}(n)$ | $\text{string}^{\S}(n)$ | length |
|---|---|---|---|---|---|
| 1 | 1 | 0 | $\varepsilon$ | $0^0 1 \varepsilon\varepsilon = 1$ | 1 |
| 2 | 10 | 1010 | 0 | 0100 | 4 |
| 3 | 11 | 1011 | 1 | 0101 | 4 |
| 4 | 100 | 10000 | 00 | 01100 | 5 |
| 5 | 101 | 10001 | 01 | 01101 | 5 |
| 6 | 110 | 10010 | 10 | 01110 | 5 |
| 7 | 111 | 10011 | 11 | 01111 | 5 |
| 8 | 1000 | 11011000 | 000 | 00100000 | 8 |

$$\text{bin}^{\#}(n) = 1^{|\text{string}(|\text{string}(n)|+1)|} \cdot 0 \cdot \overline{\text{string}(|\text{string}(n)| + 1)} \cdot \text{string}(n),$$

and the pair $(i, \text{string}(j))$ is coded by $\text{bin}^{\#}(i+1) \cdot \text{string}^{\S}(j+1)$, obtaining a string of length $2 \cdot \text{Log}(\text{Log}(i+1)+1) + \text{Log}(i+1) + 2 \cdot \text{Log}(\text{Log}(j+1)+1) + \text{Log}(j+1) + 2 < 2(\text{Log}(i) + \text{Log}(j)) + 4$ almost everywhere.

By iterating the formula (1.4) we can indefinitely improve almost everywhere the encoding of the pairs $(i, \text{string}(j))$ obtaining more and more efficient variants of Theorem 9.

**Theorem 13.** *We can construct a sequence of computable sets $(S_n)_{n\geq 1}$ such that:*

*a) for every $\sigma \in S_n$, $T_\sigma^{S_n}$ is a transducer,*

*b) for every transducer $T$ one can compute a code $\sigma \in S_n$ such that $T = T_\sigma^{S_n}$,*

*c) the difference in length between the encodings of the pair $(i, string(j))$ according to $S_n$ and $S_{n+1}$ tends to $\infty$ with $n$.*

*Proof.* This is proven in a similar fashion to the proof of Theorem 9. □

## 1.4 Summary

In this chapter, we covered the background needed for this thesis, introducing the concepts in automata and computability theory which brought us to this study. Furthermore, we presented in detail what we mean by a transducer and proffered our main results concerning transducers, in particular our encodings and hence our means of enumerating these transducers as a regular set of binary strings.

Chapter 2

# Finite-State Complexity

In this chapter we present the finite-state complexity, first proposed as an analogue of Kolmogorov complexity in [33] then in a more refined version in [12]. First we define the complexity measure in Section 2.1. Then, in the same section we present its main theoretical results. As mentioned briefly in Chapter 1, the finite-state complexity is computable. We prove this property in Section 2.2 where we also explore and begin to exploit it. In Section 2.3 we deepen the study of this complexity measure and present its main computational results.

## 2.1 Proposing an Analogue of Kolmogorov Complexity

### 2.1.1 Definition

Transducers, as previously defined, are used to 'define' or 'represent' strings in the following way. First we fix a computable set $S$ of encodings as in Theorem 9 or Theorem 13. Then we say that a pair $(T_\sigma^S, p)$, $\sigma \in S, p \in \Sigma^*$, defines the string $x$ provided $T_\sigma^S(p) = x$; the pair $(T_\sigma^S, p)$ is called a *description* of $x$. We define the size of the description $(T_\sigma^S, p)$ of $x$ by

$$||(T_\sigma^S, p)|| = |\sigma| + |p|.$$

Based on the above and on the fact that we are proposing an analogue version of Kolmogorov complexity, we define the finite-state complexity of a string as the length of the shortest description of that string, according to transducers. Formally, the definition is as follows.

**Definition 14.** The *finite-state complexity (with respect to the enumeration $S$)* of a string $x \in \Sigma^*$ is

$$\begin{aligned} C_S(x) &= \inf_{\sigma \in S,\, p \in \Sigma^*} \left\{ ||(T_\sigma^S, p)|| \ : T_\sigma^S(p) = x \right\} \\ &= \inf_{\sigma \in S,\, p \in \Sigma^*} \left\{ \mid \sigma \mid + \mid p \mid : T_\sigma^S(p) = x \right\}. \end{aligned}$$

How 'objective' is the above definition? Firstly, finite-state complexity depends on the enumeration $S$; if $S$ and $S'$ are encodings then $C_{S'} = f(C_S)$, for some computable function $f$.

Secondly, finite-state complexity is defined as an analogue of the complexity used in AIT whose objectivity is given by the Invariance Theorem which in turn relies essentially on the Universality Theorem [10]. Using the existence of a universal (prefix-free) Turing machine one can obtain a complexity which is optimal up to an additive constant (the constant 'encapsulates' the size of this universal machine). For this reason the complexity does not need to explicitly include the size of the universal machine. In sharp contrast, the finite-state complexity has to count the size of the transducer as part of the encoding length[1] but can be more relaxed in working with the pair $(\sigma, p)$. The reason is that there is no 'universal' transducer. However, we still have a strong invariance theorem which relates the 'generalised' finite-state complexity to its 'specialised' (or more restricted) complexities, analogous to AIT's Invariance Theorem.

Thirdly, our proposal does not define just one finite-state complexity but rather a class of 'finite-state complexities' (depending on the underlying enumeration of transducers). At this stage we do not have a reasonable 'invariance' result relating every pair of complexities in this class. In the theory of left-computable $\varepsilon$–randomness [11] the difference between two prefix complexities induced by different $\varepsilon$–universal prefix-free Turing machines can be arbitrarily large. In the same way here it is possible to construct two enumerations $S'$ and $S''$ satisfying Theorem 13 such that the difference between $C_{S'}$ and $C_{S''}$ is arbitrarily large.

### 2.1.2 Universality and Invariance

Below we establish in a slightly more general fashion that no finite generalised transducer can simulate a transducer on a given input—not an unexpected result. For this we note the following two lemmas, and also that the pair $(\sigma, w)$ can be uniquely encoded into the string $\sigma^\dagger w$.

**Lemma 15.** ([6], Corollary 6.2) *Any rational relation can be realised by a transducer where the transitions are a subset of $Q \times (X \cup \{\varepsilon\}) \times (Y \cup \{\varepsilon\}) \times Q$.*

[1] One can also use this approach in AIT [40], as we have also briefly mentioned in Section 1.2.

**Lemma 16.** *For any functional generalised transducer $T$ there exists a constant $M_T$ such that every prefix of an accepting computation of $T$ that processes input $x \in \Sigma^+$ produces an output of length at most $M_T \cdot |x|$.*

*Proof.* The statement follows from the observation that no functional generalised transducer can have a cycle where all transitions have input label $\varepsilon$. $\square$

**Theorem 17.** *Let $S$ be an enumeration satisfying Theorem 9.*[2] *There is no functional generalised transducer $U$ such that for all $\sigma \in S$ and $w \in \Sigma^*$, $U(\sigma^\dagger w) = T^S_\sigma(w)$.*

*Proof.* For the sake of contradiction assume that $U$ exists and without loss of generality we assume that the transitions of $U$ are in the normal form of Lemma 15. Let $M_U$ be the corresponding constant given by Lemma 16.

Let $\sigma_i \in S$, $i \geq 1$, be the encoding of the single-state transducer where the two self-loops are labeled by $0/0^i$ and $1/\varepsilon$, i.e. $\Delta(1, 0) = (1, 0^i), \Delta(1, 1) = (1, \varepsilon)$.

Define the function $g : \mathbb{N} \to \mathbb{N}$ by setting

$$g(i) = |\sigma_i^\dagger| \cdot M_U + 1, \quad i \geq 1.$$

Let $D_i$ be a valid computation of $U$ that corresponds to the input ${\sigma_i}^\dagger \cdot 0^{g(i)}$, $i \geq 1$. Let $q_i$ be the state of $U$ that occurs in the computation $D_i$ immediately after consuming the prefix ${\sigma_i}^\dagger$ of the input. Since $U$ is in the normal form of Lemma 15, $q_i$ is defined.

Choose $j < k$ such that $q_j = q_k$. We consider the computation $D$ of $U$ on input $\sigma_j^\dagger \cdot 0^{g(k)}$ that reads the prefix $\sigma_j^\dagger$ as $D_j$ and the suffix $0^{g(k)}$ as $D_k$. Since $q_j = q_k$ this is a valid computation of $U$ ending in an accepting state.

On prefix $\sigma_k^\dagger$ the computation $D_k$ produces an output of length at most $M_U \cdot |\sigma_k^\dagger|$ and, hence, on the suffix $0^{g(k)}$ the computation $D_k$ (and $D$) outputs $0^z$ where

$$z \geq k \cdot g(k) - |\sigma_k^\dagger| \cdot M_U > (k-1) \cdot g(k).$$

The last inequality follows from the definition of the function $g$. Hence the output produced by the computation $D$ is longer than $j \cdot g(k) = |T^S_{\sigma_j}(0^{g(k)})|$ and $U$ does not simulate $T^S_{\sigma_j}$ correctly. $\square$

In the remainder of this section we fix an enumeration $S$ satisfying Theorem 13.

**Proposition 18.** *For every pair of strings $x, y \in \Sigma^*$ there exist infinitely many transducers $T^S_\sigma$ such that $T^S_\sigma(x) = y$.*

[2]We use a regular enumeration to avoid the possibility that the non-existence of a universal transducer is simply caused by the fact that a finite transducer cannot recognise legal encodings of transducers.

*Proof.* Given $x, y \in \Sigma^*$, with $x = x_1x_2 \ldots x_n$ of length $n$, we construct the transducer $T_\sigma = (Q, 1, \Delta)$ having $n+1$ states acting as follows: $\Delta(i, x_i) = (n+1, \varepsilon)$, $1 \leq i \leq n$, $\Delta(1, x_1) = (2, y)$, $\Delta(j, x_j) = (j+1, \varepsilon)$, $2 \leq j \leq n$, $\Delta(n+1, 0) = (n+1, \varepsilon)$. □

In spite of the negative result stated in Theorem 17, the Invariance Theorem from AIT is true for $C$. To this aim we define the complexity associated with a transducer $T^S_\sigma$ by

**Definition 19.** The *finite-state complexity (with respect to a fixed transducer $T^S_\sigma$) of a string $x \in \Sigma^*$ is*

$$C_{T^S_\sigma}(x) = \inf_{p \in \{0,1\}^*} \Big\{ \mid p \mid : T^S_\sigma(p) = x \Big\}.$$

**Theorem 20.** (Invariance) *For every $\sigma_0 \in S$ we have $C_S(x) \leq C_{T^S_{\sigma_0}}(x) + |\sigma_0|$, for all $x \in \Sigma^*$.*

*Proof.* Using the definitions of $C_S$ and $C_{T^S_{\sigma_0}}$ we have:

$$\begin{aligned} C_S(x) &= \inf_{\sigma \in S,\ p \in \Sigma^*} \Big\{ \|(T^S_\sigma, p)\| : T^S_\sigma(p) = x \Big\} \\ &= \inf_{\sigma \in S,\ p \in \Sigma^*} \Big\{ |\sigma| + |p| : T^S_\sigma(p) = x \Big\} \\ &\leq |\sigma_0| + \inf_{p \in \Sigma^*} \Big\{ |p| : T^S_{\sigma_0}(p) = x \Big\} \\ &= C_{T^S_{\sigma_0}}(x) + |\sigma_0|. \end{aligned}$$

□

**Corollary 21.** *If $T^S_{\sigma_0}(x) = x$, then $C_S(x) \leq |x| + |\sigma_0|$, for all $x \in \Sigma^*$. In particular, using Example 12 (last transducer) we deduce that $C_{S_0}(x) \leq |x| + 8$, for all $x \in \Sigma^*$.*

**Corollary 22.** *The complexity $C_S$ is computable.*

**Conjecture 23.** *The computational complexity of testing whether the finite-state complexity of a string $x$ is less or equal to $n$ is in NP.*

**Open Problem 24.** *Is the decision problem expressed in Conjecture 23 NP-hard?* (See also [5, 28].)

The implications of these statements lie in the domain of efficiency when we deal with computing the finite-state complexity of a string. Assuming that P $\neq$ NP, then given a string $x$ and an integer $n$, if testing whether $C_S(x) \leq n$ is NP-hard, then by implication there is no efficient way to compute the finite-state complexity. We conjecture that this decision problem is in NP, but we highly suspect that it is the case that it is in fact NP-hard. However, proving this would be extremely difficult.

### 2.1.3 Quantitative Estimates

Here we establish basic upper and lower bounds for finite-state complexity of arbitrary strings as well as for strings of particular types. For the rest of this section we use the enumeration $S_0$. Furthermore, we write $T_\sigma$ and $C$ instead of $T_\sigma^{S_0}$ and $C_{S_0}$.

**Definition 25.** *Let $f$ and $g$ be two functions. We say that $f(n) \in \Theta(g(n))$ if and only if $c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$ for all $n > n_0$, where $c_1, c_2$ are positive constants.*

**Theorem 26.** *For $n \ge 1$ we have: $C(0^n) \in \Theta(\sqrt{n})$.*

*Proof.* It is sufficient to establish that

$$2 \cdot \lfloor \sqrt{n} \rfloor \le C(0^n) \le 4 \cdot \lfloor \sqrt{n} \rfloor + \alpha, \tag{2.1}$$

where $\alpha$ is a constant.

For the upper bound we note that $0^n$ can be represented by a pair $(T, p)$ where $T$ is a single state transducer having two self-loops labeled respectively, $0/0^{\lfloor \sqrt{n} \rfloor}$ and $1/0$, and $p$ can be chosen as a string $0^{\lfloor \sqrt{n} \rfloor + y} 1^z$, where $0 \le y \le 1$, $0 \le z \le \lfloor \sqrt{n} \rfloor$. By our encoding conventions the size of $(T, p)$ is at most $4 \cdot \lfloor \sqrt{n} \rfloor + \alpha$ where $\alpha$ is a small constant.

To establish the lower bound, consider an arbitrary pair $(T', p')$ representing $0^n$. If $v$ is the longest output of any transition of $T'$, then $|v| \cdot |p'| \ge n$. On the other hand, according to our encoding conventions $||(T', p')|| \ge 2 \cdot |v| + |p'|$. These inequalities imply $||(T', p')|| \ge 2 \cdot \lfloor \sqrt{n} \rfloor$. □

Using a more detailed analysis the upper and lower bounds of (2.1) could be moved closer to each other. But, because the precise multiplicative constants depend on the particular enumeration $S_0$, it should not be very important to try to improve the values of the multiplicative constants.

The argument used to establish the lower bound in (2.1) directly provides the following:

**Corollary 27.** *For any $x \in \Sigma^*$, $C(x) \ge 2 \cdot \lfloor \sqrt{|x|} \rfloor$.*

The bounds (2.1) imply that the inequality $H(xx) \le H(x) + O(1)$ familiar for Kolmogorov complexity does not hold for finite-state complexity:

**Corollary 28.** *There is no constant $\alpha$ such that for all strings $x \in \Sigma^*$, $C(xx) \le C(x) + \alpha$.*

The result in Corollary 28 implies that, contrary to intuition, as far as finite-state complexity is concerned the string $xx$ contains significantly more information than the string $x$.

The mapping $0^n \mapsto 0^{2\cdot n}$ is computed by a transducer of small size. Hence we deduce:

**Corollary 29.** *For a given transducer $T$ there is no constant $\alpha$ such that for all strings $x \in \Sigma^*$, $C(T(x)) \leq C(x) + \alpha$.*

In Corollary 29 we require only that $\alpha$ is independent of $x$. That is the value $\alpha$ could depend on the transducer $T$. This result implies that the pair describing $T(x)$ for a given transducer $T$ on any string $x$ will always hold more information than the string $x$ itself. As in Theorem 26 we get estimations for the finite-state complexity of powers of a string.

**Proposition 30.** *For $u \in \Sigma^*$ and $n \gg |u|$,*

$$C(u^n) \leq 2 \cdot (\lfloor \sqrt{n} \rfloor + 1) \cdot |u| + 2\lfloor \sqrt{n} \rfloor + \alpha, \tag{2.2}$$

*where $\alpha$ is a constant independent of $u$ and $n$.*

*Proof.* Let $T$ be the single state transducer with two self-loops labeled respectively by $0/u^{\lfloor \sqrt{n} \rfloor}$ and $1/u$. The string $u^n$ has a description $(T, 0^{\lfloor \sqrt{n} \rfloor + y} 1^z)$ where $0 \leq y \leq 1$, $0 \leq z \leq \lfloor \sqrt{n} \rfloor$. By our encoding conventions

$$||(T, 0^{\lfloor \sqrt{n} \rfloor} 1^z)|| = 2 \cdot (\lfloor \sqrt{n} \rfloor + 1) \cdot |u| + 4 + \lfloor \sqrt{n} \rfloor + y + z.$$

Note that, when encoding self-loops, the state name is not part of the encoding and a self-loop with output string $w$ contributes $2|w| + 2$ to the length of the encoding. The claim follows from the upper bounds for $y$ and $z$. □

The upper bound (2.2) is useful only when $n$ is larger than $|u|^2$ because using a single state transducer with self-loop $0/u$ we get an upper bound $C(u^n) \leq 2 \cdot |u| + n + \alpha$, with $\alpha$ constant.

**Corollary 31.** *We have: $C(0^n 1^n) \in \Theta(\sqrt{n})$.*

*Proof.* The lower bound follows from Corollary 27. The string $0^n 1^n$ has description

$$(T, 0^{\lceil \sqrt{n} \rceil - 1 + y_1} 1^{z_1} 0^{z_2} 1^{\lceil \sqrt{n} \rceil - 1 + y_2}),$$

where $0 \leq y_1, y_2 \leq 1$, $1 \leq z_1, z_2 \leq \lceil \sqrt{n} \rceil$ and $T$ is the transducer given in Figure 2.1.

Note that, differing from the construction used in Theorem 26, the transducer in Figure 2.1 begins by outputting strings $0^{\lceil \sqrt{n} \rceil - 1}$ (instead of $0^{\lfloor \sqrt{n} \rfloor}$). This is done in order to guarantee that $z_1$ can be chosen to be at least 1 also when $n$ is a perfect square.

Thus $C(0^n 1^n) \leq 8 \cdot \lceil \sqrt{n} \rceil + \alpha$ where $\alpha$ is a constant. □

From Corollary 31 we note that the finite-state complexity of $0^n1^n$ is within a constant factor of the automatic complexity (as defined in [39]) of the same string. This can be viewed merely as a coincidence since the two descriptional complexity measures are essentially different and generally have very different upper and lower bounds.

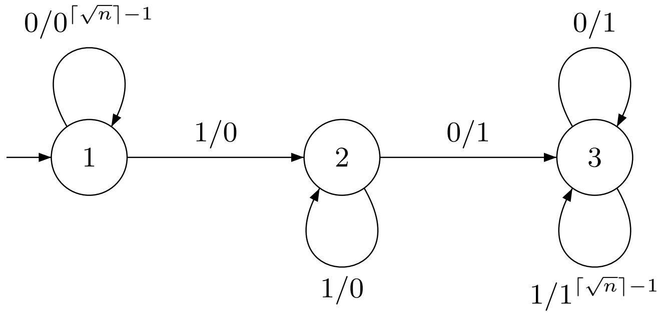
$0/0^{\lceil\sqrt{n}\rceil-1}$
$0/1$
$1/0$
$0/1$
1
2
3
$1/0$
$1/1^{\lceil\sqrt{n}\rceil-1}$

Figure 2.1: Transducer $T$ in the proof of Corollary 31.

The following result gives an upper bound for finite-state complexity of the catenation of two strings.

**Proposition 32.** *For any $\omega > 0$ there exists $d(\omega) > 0$ such that for all $x, y \in \Sigma^*$,*

$$C(xy) \leq (1+\omega)\cdot(4C(x)+C(y))+d(\omega).$$

*Here the value $d(\omega)$ depends only on $\omega$, i.e., it is independent of $x$ and $y$.*

*Proof.* Let $(T, u)$ and $(R, v)$ be minimal descriptions of $x$ and $y$, respectively. Let $u = u_1 \cdots u_m$, $u_i \in \Sigma$, $i = 1, \ldots, m$ and recall that $u^\dagger = u_1 0 u_2 0 \cdots u_{m-1} 0 u_m 1$.

Denote the sets of states of $T$ and $R$, respectively, as $Q_T$ and $Q_R$, and let $Q'_T = \{q' \mid q \in Q_T\}$.

We construct a transducer $W$ with set of states $Q_T \cup Q'_T \cup Q_R$ as follows.

1. For each transition of $T$ from state $p$ to state $q$ labeled by $i/w$ ($i \in \Sigma$, $w \in \Sigma^*$), $W$ has a transition from $q$ to $p'$ labeled by $i/w$ and a transition labeled $0/\varepsilon$ from $p'$ to $p$.

2. Each state $p' \in Q'_T$ has a transition labeled $1/\varepsilon$ to the starting state of $R$.

3. The transitions originating from states of $Q_R$ are defined in $W$ in the same way as in $R$.

Now $|u^\dagger| = 2 \cdot |u|$ and

$$W(u^\dagger v) = T(u)R(v) = xy.$$

It remains to verify that the size of the encoding of $W$ is roughly at most four times the size of $T$ plus the size of $R$.

First assume that

**(A)** the states of $W$ could have the same length encodings as the encodings used for states in $T$ and $R$.

We note that the part of $W$ simulating the computation of $T$ has simply doubled the number of states and for the new states of $Q'_T$ the outgoing transitions have edge labels of minimal length ($0/\varepsilon$ and $1/\varepsilon$). An additional increase in the length of the encoding occurs because each self-loop of $T$ is replaced in $W$ by two transitions that are not self-loops. It is easy to establish, using induction on the number of states of $T$, that if all states of $T$ are reachable from the start state and $T$ has $t$ non-self-loop transitions, the number of self-loops in $T$ is at most $t+2$.

Thus by the above observations with the assumption (A), $C(xy)$ could be bounded above by $4C(x)+C(y)+d$ where $d$ is a constant. Naturally, in reality the encodings of states of $W$ need one or two additional bits added to the encodings of the corresponding states in $T$ and $R$. The proportional increase of the state encoding length caused by the two additional bits for the states of $Q_T \cup Q'_T$, (respectively, states of $Q_R$) is bounded above by $2\cdot(\lceil\log_2(|Q_T|)\rceil)^{-1}$ (respectively, $2\cdot(\lceil\log_2(|Q_R|)\rceil)^{-1}$). Thus, the proportional increase of the encoding length becomes smaller than any positive $\omega$ when $\max\{|Q_T|, |Q_R|\}$ is greater than a suitably chosen threshold $M(\omega)$. On the other hand, the encoding of $W$ contains at most $2\cdot(2|Q_T|+|Q_R|) \leq 6\cdot\max\{|Q_T|, |Q_R|\}$ occurrences of substrings encoding the states. This means that by choosing $d(\omega) = 12 \cdot M(\omega)$ the statement of the lemma holds also for small values of $|Q_T|$ and $|Q_R|$. □

The proof of Proposition 32 relies on an estimation that the part of the transducer $W$ simulating the computation of $R$ has an encoding at most four times the size of the encoding of $R$. The additional increase is caused by the complication that each self-loop is simulated by two non-self-loops and the encoding of transitions that are not self-loops needs to include the state names. Using a more detailed analysis the constant 4 could likely be improved.

It is known that deterministic transducers are closed under composition [6]. That is, for transducers $T_\delta$ and $T_\gamma$ there exists $\sigma \in S$ such that $T_\sigma(x) = T_\delta(T_\gamma(x))$ for all $x \in \Sigma^*$. Using the construction from [6] (Proposition 2.5, page 101) we give an upper bound for $|\sigma|$ as a function of $|\delta|$ and $|\gamma|$.

Let $T_\delta = (Q, q_0, \Delta)$ and $T_\gamma = (P, p_0, \Gamma)$, where $\Delta$ is a function $Q \times \Sigma \to Q \times \Sigma^*$ and $\Gamma$ is a function $P \times \Sigma \to P \times \Sigma^*$. The transition function $\Delta$ is extended in the natural way as a function $\hat{\Delta} : Q \times \Sigma^* \to Q \times \Sigma^*$.

The composition of $T_\gamma$ and $T_\delta$ is computed by a transducer $T_\sigma = (Q \times P, (q_0, p_0), \Xi)$ where $\Xi : Q \times P \times \Sigma \to Q \times P \times \Sigma^*$ is defined by setting for $q \in Q$, $p \in P$, $a \in \Sigma$,

$$\begin{aligned}&\Xi\left(\left(q,p\right),a\right) = \\ &\left(\left(\pi_1\left(\hat{\Delta}\left(q,\pi_2\left(\Gamma\left(p,a\right)\right)\right)\right),\pi_1\left(\Gamma\left(p,a\right)\right)\right),\pi_2\left(\hat{\Delta}\left(q,\pi_2\left(\Gamma\left(p,a\right)\right)\right)\right)\right).\end{aligned}$$

The number of states of $T_\sigma$ is bounded above by $|\delta| \cdot |\gamma|$.[3] An individual output of $T_\sigma$ consists of the output produced by $T_\delta$ when it reads an output produced by one transition of $T_\gamma$ (via the extended function $\hat{\Delta}$). Therefore the length of the output produced by an individual transition of $T_\sigma$ can be bounded above by $|\delta| \cdot |\gamma|$. These observations imply that

$$|\sigma| = O(|\delta|^2 \cdot |\gamma|^2).$$

The above estimate was obtained simply by combining the worst-case upper bound for the size of the encoding of the states of $T_\sigma$ and the worst-case length of individual outputs of the transducers $T_\delta$ and $T_\gamma$. The worst-case examples for these two bounds are naturally very different as the latter corresponds to a situation where the encoding of individual outputs 'contributes' a large part of the strings $\delta$ and $\gamma$. The overall upper bound could be somewhat improved using a more detailed analysis.

**Open Problem 33.** *Is it possible to obtain a reasonable upper bound for $C(u)$ in terms of $C(v)$ when $u$ is a prefix of $v$?*

### 2.1.4 Incompressibility and Lower Bounds

Following the model of incompressibility in AIT we obtain:

**Definition 34.** A string $x$ is *finite-state $i$–compressible* ($i \geq 1$) if $C(x) \leq |x| - i$.

**Definition 35.** A string $x$ is *finite-state $i$–incompressible* ($i \geq 1$) if $C(x) > |x| - i$. If $i = 1$, then the string is called *finite-state incompressible.*

**Lemma 36.** *There exist finite-state incompressible strings of any length.*

[3]Strictly speaking this could be multiplied by $\frac{(\log_2 \log_2 |\delta|)\cdot(\log_2 \log_2 |\gamma|)}{\log_2 |\delta| \cdot \log_2 |\gamma|}$ to give a better estimate.

*Proof.* We simply note that the set $\{x : |x| = n, C(x) \leq |x| - i\}$ has at most $2^{n-i+1} - 1$ elements. □

Lemma 36 relies on a standard counting argument and does not give a construction of incompressible strings. By relying on results on grammar-based compression we can get lower bounds for finite-state complexity of explicitly constructed strings.

A grammar $G$ (or straight-line program [15, 23, 29, 34]) used as an encoding of a string has a unique production for each nonterminal. Furthermore, the grammar is acyclic as defined in Section 1.2. That is, there is an ordering of the nonterminals $X_1, \ldots, X_m$ such that the productions are of the form $X_1 \to \alpha_1, \ldots, X_m \to \alpha_m$, where $\alpha_i$ contains only nonterminals from $\{X_{i+1}, \ldots, X_m\}$ and terminal symbols.

**Definition 37.** The *size of the grammar $G$*, $\text{size}(G)$, is $\sum_{i=1}^{m} |\alpha_i|$.

Grammar-based compression of a string $x$ may result in exponential savings compared to the length of $x$. Comparing this to Corollary 27 we note that the size of the smallest grammar generating a given string may be exponentially smaller than the finite-state complexity of the string. Conversely, any string $x$ can be generated by a grammar with size $O(C(x))$.

**Lemma 38.** *There exists a constant $d \geq 1$ such that for any $x \in \Sigma^*$, $\{x\}$ is generated by a grammar $G_x$ where $\text{size}(G_x) \leq d \cdot C(x)$.*

*Proof.* The construction outlined in [20] for simulating an 'NFA with advice' by a grammar is similar. For the sake of completeness we include here a construction.

Assume $x$ is encoded as a transducer-string pair $(T_\sigma, p)$, where $p = p_1 \cdots p_n$, $p_i \in \Sigma$. The initial nonterminal of the grammar $G_x$ has a production with right side $(p_1, s_{i_1})(p_2, s_{i_2}) \cdots (p_n, s_{i_n})$ where $s_{i_j}$ is the state of $T_\sigma$ reached by the transducer after consuming the input string $p_1 \cdots p_{j-1}$, $1 \leq j \leq n$. After this the rules for nonterminals $(p_i, s)$ simply simulate the output produced by $T_\sigma$ in state $s$ on input $p_i$.

Let $Q$ be the set of states of $T_\sigma$ and, as usual, denote the set of transitions by $\Delta : Q \times \Sigma \to Q \times \Sigma^*$. The size of $G_x$, that is the sum of the lengths of right sides of the productions of $G_x$, is

$$\text{size}(G_x) = |p| + \sum_{q \in Q, i \in \Sigma} |\pi_2(q, i)|.$$

□

Note that $\text{size}(G_x)$ is defined simply as the sum of lengths of productions of $G_x$ while $C(x)$ uses a binary encoding of a transducer $T$. In cases where minimal representations use transducers with fixed length outputs for individual transitions and large numbers of states, size$(G_x)$ is less than a constant times $C(x) \cdot (\log_2 C(x))^{-1}$.

Next we show some interesting results on the finite-state complexity of specific types of words: *de Bruijin words*. These results show that these information-filled words will always have high finite-state complexity. A binary de Bruijn word of order $r \geq 1$ is a string $w$ of length $2^r + r - 1$ over alphabet $\Sigma$ such that any binary string of length $r$ occurs as a substring of $w$ (exactly once). It is well known that de Bruijn words of any order exist and have an explicit construction [22, 43].

**Theorem 39.** *There is a constant $d$ such that for any $r \geq 1$ there exist strings $w$ of length $2^r + r - 1$ with an explicit construction such that $C(w) \geq d \cdot |w| \cdot (\log_2(|w|))^{-1}$.*

*Proof.* It is known that any grammar generating a de Bruijn string of order $r$ has size $\Omega(\frac{2^r}{r})$ [3]. Grammars generating a singleton language are called string chains in [3]. See also [23]. The claim follows by Lemma 38. $\square$

**Conjecture 40.** *de Bruijn words are finite-state incompressible.*

We therefore have a specific construction of words which will always yield a high complexity measure, a set that we suspect is in fact finite-state incompressible, as stated in Conjecture 40. These sorts of results doubly increase the interest in the computability of this complexity measure and developing an effective and efficient algorithm in order to empirically explore its results becomes an inevitability.

## 2.2 Computing the Finite State Complexity

Since we have a computable complexity measure (see Corollary 22), it is natural to attempt to find an efficient algorithm to compute the finite-state complexity. This subsection presents the algorithm's pseudocode along with the many approaches that were explored to compute both our $S_0$ and $S_1$ encodings. We also present the optimisation features that we implemented. Finally, we give some initial complexity results obtained from the computations.

### 2.2.1 One Algorithm to Compute Them All

Below is the general pseudocode outlining all the programs we devised to compute the finite-state complexity. We found that even though certain details are computed differently and some constant values change from one encoding to another, the general approach remains the same. This is supported by Conjecture 23. No matter what clever tricks we may think of to better the efficiency, the overall algorithm always fundamentally employs a brute force approach.

The general idea of the algorithm is that given an input $n$ it enumerates through all strings of length $n$ and following a specific encoding, finds all the legal transducer/input pairs within these and stores those along with their output string in an ever-growing table. More technically, given $n$, it enumerates through all possible strings $\rho$, where $\rho$ should be of the form $\sigma \cdot p$ and $|\rho| = n$. We must check for every $\rho$ whether it is of the correct form. For every successful $\rho$ we extract the pair $(T_\sigma, p)$ and test whether $T_\sigma(p)$ successfully outputs a string $x$. If so, then we can measure its complexity, $C^S_\sigma(x) = |\sigma| + |p|$ (which equals $n$). In all other cases, we reject the current $\rho$ and continue with the enumeration. The algorithm terminates once all possible $\rho$s of length $n$ have been explored; that is, from $\underbrace{00\ldots0}_{n}$ to $\underbrace{11\ldots1}_{n}$.

The table, or database, that this algorithm populates aims at holding all possible finite-state complexities. The table is populated in co-lexicographical order of the pairs, therefore in increasing 'finite-state complexity order'. Here is the algorithm below.

**Algorithm 2.2.1:** FiniteStateComplexity(int $n$)

```
procedure FSComplexity(int $n$)
 $upperBd \leftarrow n$
 for each string enumeration $\rho$ such that $|\rho| == upperBd$
   do { if correctEncoding($\rho$) and obtainsX($T_\sigma, p$)
          then return ( $|\rho|$ )

procedure correctEncoding(String $\rho$)
 if $\rho$ has correct pattern
   then { $\sigma \leftarrow$ prefix of $\rho$
          $p \leftarrow$ rest of $\rho$
          comment: find and assign each bin($i$) and $u_i$ to their array
          for each bin($i$) $\in \sigma$ and $u_i \in \sigma$ (i.e in $T_\sigma$)
            do { $bins \leftarrow$ bin($i$)
                 $outs \leftarrow u_i$
          if $|bins| = |outs|$ and $|bins|$ is even and $|bins| > 0$
            then return ( true )
 return ( false )
```

```
procedure OBTAINSX(Array $bins$, Array $outs$, String $p$)
 for each bin($i$) $\in bins$
   do { if $i = 0$ or $i \geq \frac{|bins|}{2}$
          then return ( false )
        $trans \leftarrow i$ (where $trans$ is the transition array)
 $output \leftarrow \varepsilon$
 $k \leftarrow 0$
 for each character $p_i \in p$
   do { if $p_i =$ '0'
          then { $t \leftarrow$ element of $trans$ at $k$
                 $output \leftarrow output +$ element of $outs$ at $k$
          else { $t \leftarrow$ element of $trans$ at $k + 1$
                 $output \leftarrow output +$ element of $outs$ at $k + 1$
        $k \leftarrow 2 \times (t - 1)$
 return ( true )
```

We now discuss each procedure of the algorithm separately in order to give some conceptual proof of correctness.

The enumeration procedure of binary strings is performed in co-lexicographic order, for every length, starting from the description of the smallest transducer which outputs $\varepsilon$ on all inputs.

The FSCOMPLEXITY($n$) procedure is the core of the algorithm. It takes an integer $n$ as input and manages both the enumeration and the other processes. Its role is to find the shortest strings $\rho$ which are the correct encodings of corresponding pairs $(T_\sigma, p)$ that each compute some string $x$. As the enumeration is in co-lexicographic order, the length of such a pair is the complexity of the string if this string has never been encountered previously. It also limits the search of such a $\rho$, based on the upper bound for its length defined in Corollary 21, and fixed by the input $n$.

The CORRECTENCODING($\rho$) procedure first checks whether the enumerated $\rho$ (passed as input to this procedure) is of the form $\sigma \cdot p$. In the affirmative it decodes $\rho$ into $\sigma$ and $p$ and then checks whether these components have the correct form and extracts $T_\sigma$ from $\sigma$. Finally, if still in the affirmative this procedure extracts the transition and the output functions from $T_\sigma$ and checks whether the size of their ranges are equal and even. If all of these conditions hold the enumerated $\rho$ is a (seemingly —as we still need to test whether the pair can successfully outputs a string) correct encoding, i.e. $\rho = \sigma \cdot p$ where $\sigma \in S$. Otherwise the procedure rejects the current $\rho$, returning to the FSCOMPLEXITY($n$) procedure to test the next possible $\rho$.

The final procedure, OBTAINSX($bins, outs, p$), is only called if we have a potential pair $(T_\sigma, p)$. It uses the extracted functions (passed as inputs to the procedure along with $p$) to simulate the computation of the transducer $T_\sigma$ on the input $p$ and stores the output. If the simulation completes then there were no errors in matching the functions representing the transducer and $\textsc{obtainsX}(bins, outs, p)$ returns true. Otherwise it returns false. The simulation may still encounter errors as the building of the functions in $\textsc{correctEncoding}(\rho)$ does not check that the functions correctly map to each other (i.e. no non-existent states are called), a check which is done here. If this procedure in fact returns true, we have found a description $(T_\sigma, p)$ for some $x$. If this $x$ was never encountered then we actually have a *minimal description*.

It is clear that since we enumerate through all possible binary strings of a fixed length, we loop over all lengths starting from the size of the description of the smallest $\sigma$ in $S$. Hence, every possible pair will eventually be enumerated exactly once. If any one of the steps in the 'decoding' fails, the algorithm rejects the considered $\rho$ and enumerates the next string in co-lexicographic order. Thus for every string $x$, the algorithm will find a minimal description; computing the finite-state complexity of $x$. Furthermore, it will halt since there are a finite amount of strings of length $n$ to consider.

Where each of the specific algorithms differ is in the handling of the encodings, which occurs in $\textsc{correctEncoding}(\rho)$ and $\textsc{obtainsX}(bins, outs, p)$. Some specifics of $\textsc{FSComplexity}(n)$ can be handled in different ways, especially the enumeration. These details are where we can work on optimising the algorithm.

### 2.2.2 Optimisations

First, we recall the fact that this algorithm is used to build a table (or database) of all the possible complexities. This table is therefore infinite in size. The longest procedure in this algorithm is the enumeration, which has to sort through all the strings and separate the 'correct' encodings from the 'incorrect' ones; as a result there is a lot of 'wasted' work. Hence we have focused on making the enumeration as efficient as possible by limiting the wasteful computations as much as we could.

The first improvement is that instead of computing the complexity of a given string directly as in [33], we have altered the algorithm to build a table (which we store in a database). Now the process of computing a specific finite-state complexity can be done in two steps. First, searching through the table for that string. Then returning its complexity if it has been computed, or starting the search from the last (largest) computed pair $(T_\sigma, p)$.

Secondly, we have explored two new possible ways for the enumeration to take form. On the one hand we let the algorithm enumerate directly over all strings $\rho$ in co-lexicographic order, as it originally did. We parsed each $\rho$, extracting all possible pairs from it instead of one. This is more clearly represented mathematically as such: Let $\rho = \sigma \cdot p$ and $|\rho| = n$. Now let $|\sigma| = k$, where $k = c$, $c + 1$,

$c+2, \cdots, n$ and $c$ is the size of the smallest transducer (as defined in Example 11) with the corresponding encoding. Then $p$ is the remainder of the string $\rho$. On the other hand we had the algorithm enumerate over all $\sigma$'s and then over all $p$'s (for each $\sigma$), both in co-lexicographic order. Hence we are dealing with two separate enumerations: one for $\sigma$, another for $p$. However, we maintain the condition that $|\rho| = n$. Therefore, in both cases we are building the table in sets, where each set comprises of all the strings of complexity some fixed $n$.

Thirdly, we developed a parallel (or distributive) approach. As we wanted to build as much of the table as possible and as fast as possible, we exploited the independent aspect of the algorithm to run several copies over a cluster of computers in parallel, building the one static database. Unfortunately, a combination of technical issues and a lack of resources severely stalled our endeavour, and it is still in the works. Nevertheless, it is currently being applied and is continuously populating our table.

### 2.2.3 Some Results

In Tables 2.1 and 2.2 we present a few initial values of $C_{S_0}$ and $C_{S_1}$, respectively. 'Complementary' strings are omitted. A plot containing the values of $C_{S_0}$ and $C_{S_1}$ appears in Figure A.1, and a more comprehensive plot is found in Figure A.2, both found in Appendix A.

Table 2.1: Finite-state complexity (w.r.t. $S_0$) of all strings in co-lexicographic order from $\varepsilon$ to 01111.

| $x$ | $C_{S_0}(x)$ | $(\sigma, p)$ | $x$ | $C_{S_0}(x)$ | $(\sigma, p)$ |
|---|---|---|---|---|---|
| $\varepsilon$ | 4 | (0000,$\varepsilon$) | 00000 | 11 | (000110,11111) |
| 0 | 7 | (000110,1) | 00001 | 13 | (01000110,11110) |
| 00 | 8 | (000110,11) | 00010 | 13 | (01000110,11101) |
| 01 | 9 | (00011100,1) | 00011 | 13 | (01000110,11100) |
| 000 | 9 | (000110,111) | 00100 | 13 | (01000110,11011) |
| 001 | 11 | (01000110,110) | 00101 | 13 | (01000110,11010) |
| 010 | 11 | (01000110,101) | 00110 | 13 | (01000110,11001) |
| 011 | 11 | (01000110,100) | 00111 | 13 | (01000110,11000) |
| 0000 | 10 | (000110,1111) | 01000 | 13 | (01000110,10111) |
| 0001 | 12 | (01000110,1110) | 01001 | 13 | (01000110,10110) |
| 0010 | 12 | (01000110,1101) | 01010 | 13 | (01000110,10101) |
| 0011 | 12 | (01000110,1100) | 01011 | 13 | (01000110,10100) |
| 0100 | 12 | (01000110,1011) | 01100 | 13 | (01000110,10011) |
| 0101 | 10 | (00011100,11) | 01101 | 13 | (01000110,10010) |
| 0110 | 12 | (01000110,1001) | 01110 | 13 | (01000110,10001) |
| 0111 | 12 | (01000110,1000) | 01111 | 13 | (01000110,10000) |

Table 2.2: Finite-state complexity (w.r.t. $S_1$) of all strings in co-lexicographic order from $\varepsilon$ to 01111.

| $x$ | $C_{S_1}(x)$ | $(\sigma, p)$ |
|---|---|---|
| $\varepsilon$ | 16 | (1010010010100100,$\varepsilon$) |
| 0 | 17 | (1010010010100101,1) |
| 00 | 18 | (1010010010100101,11) |
| 01 | 18 | (10100100101001110,1) |
| 000 | 19 | (1010010010100101,111) |
| 001 | 19 | (10100101101001110,01) |
| 010 | 19 | (10100101101001110,10) |
| 011 | 20 | (10100101101001100,011) |
| 0000 | 19 | (10100100101001101,11) |
| 0001 | 20 | (10100101101001110,001) |
| 0010 | 20 | (10100101101001110,010) |
| 0011 | 21 | (10100101101001100,0011) |
| 0100 | 20 | (10100101101001110,100) |
| 0101 | 19 | (10100100101001110,11) |
| 0110 | 20 | (101001110101001111,01) |
| 0111 | 21 | (10100101101001100,0111) |
| 00000 | 20 | (10100101101001101,011) |
| 00001 | 21 | (10100101101001110,0001) |
| 00010 | 21 | (10100101101001110,0010) |
| 00011 | 22 | (10100101101001100,00011) |
| 00100 | 21 | (10100101101001110,0100) |
| 00101 | 20 | (10100101101001110,011) |
| 00110 | 22 | (10100101101001100,00110) |
| 00111 | 22 | (10100101101001100,00111) |
| 01000 | 21 | (10100101101001110,1000) |
| 01001 | 20 | (10100101101001110,101) |
| 01010 | 20 | (10100101101001110,110) |
| 01011 | 21 | (101001100101001110,110) |
| 01100 | 22 | (10100101101001100,01100) |
| 01101 | 21 | (101001100101001110,101) |
| 01110 | 22 | (10100101101001100,01110) |
| 01111 | 22 | (10100101101001100,01111) |

Currently these tables have been completed for all strings of complexities at most 40 for the $S_0$ encoding, and at most 30 for the $S_1$ encoding. The aforementioned plots include all strings for which the complexity in both encodings were found. These plots show the overall trend of the two complexities, and offer a visual comparison of these. We can directly see that $S_1$ allows for a more compact encoding, and they heavily suggest that in the long run, for all $x \in \Sigma^*$, $C_{S_1}(x) \leq C_{S_0}(x)$.

In Subsection 2.1.4, we have mentioned Borel normality and our interest in linking the property with finite-state incompressibility (FS-incompressibility). In AIT, we have the following theorem correlating Borel normality and (algorithmic) incompressibility [9].

**Theorem 41** ([9]). *There exists an integer $M$ such that for all $x \in \Sigma^*$ and $|x| \geq M$, $x$ is incompressible implies that $x$ is Borel normal, but the converse implication is false.*

We believe that an equivalent theorem holds true for finite-state complexity. Even though we lack the theoretical results proving this, we do have partial results and some experimental evidence hinting towards it, which we present here. In the charts represented in Figure 2.2 and Figure 2.3 we show the results from comparing how many strings are FS-incompressible, and how many are Borel normal (according to Trenton's algorithm [41] which is heavily based on Calude's work [9]). We first need to note that Borel normality is an asymptotic property of strings, and strings of length greater than or equal to 16, as discussed in [41], are proven to always evaluate to being Borel normal according to the formula given in Definition 6. Therefore, in the concerned charts we ignored all strings of length strictly less than 16. Second, a similar phenomenon is true for FS-compressibility since our encodings will not truly allow for compression until we have reached strings of a certain length (in order for the description of the string to be shorter than the identity pair). In our results the first encountered FS-compressible string, in the $S_0$ encoding, is the string $w_1 = 000000000000000000000000000000000 = 0^{33}$. In fact, we obtain 76 FS-compressible strings out of 9822 strings (excluding those of length $< 16$). As we still have not encountered an FS-compressible string for the $S_1$ encoding results, we focus our Borel normal results on the $S_0$ encoding measurements.

For the $S_0$ encoding results, let us begin by stating that for strings of length $\leq 40$, 99.23% are FS-incompressible and 93.73% are Borel normal. Among those we find that most Borel normal strings are FS-incompressible (99.28%). Only a very small set of strings that are Borel normal are also FS-compressible (only 0.72%, that is 66 strings). Moreover, most FS-compressible strings are Borel normal (86.84%), surprisingly since we would expect the FS-compressible strings to be not Borel normal; this result should be taken with caution as the set concerned is very small, with only 76 strings being FS-compressible. In our data, there exists a string

$$w_2 = 001001001001001001001001001001001 = (001)^{11}$$

such that $w_2$ is FS-compressible and Borel normal. Note that $|w_2| = 33$ which is the length of the first FS-compressible string, and in fact all of the strings in this case have length greater or equal to 33. In the plot below, in Figure 2.4, we show

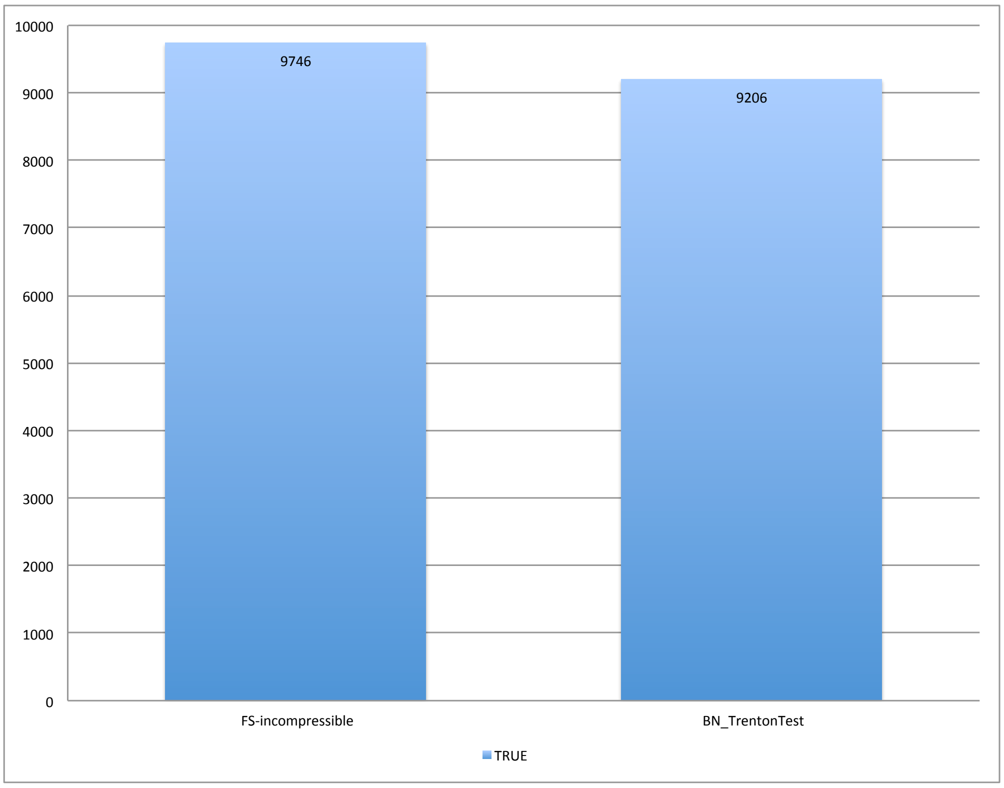

Figure 2.2: Comparing FS-incompressibility and Borel-Normality on 9822 strings of length $\geq 16$ and of FS-complexity $\leq 42$, using $S_0$.

the trend of the data that is both FS-compressible and Borel normal in terms of lengths of strings.

A clear 'step' function is apparent in Figure 2.4 which shows that the number of strings that are both FS-compressible and Borel normal increases proportionally with the number of strings of greater lengths. But we are missing data for strings of considerably greater length before we can draw stronger conclusions. Another notable fact is that the number of these strings is only 66, a number which could either continue to minimally increase (with the size of the data set) or suddenly peak, even though no evidence suggests the latter. From our example and all the data put forth in Figure 2.4, we have therefore proved the following.

**Lemma 42.** *There exists a string $x \in \Sigma^*$, $|x| = 40$ such that $x$ is both Borel normal and finite-state compressible; in particular, Borel normality does not imply FS-incompressibility for strings of length up to* 40.

Lemma 42 is a weak version of the converse statement in Theorem 41. But the evidence that we have collected so far still suggests that we can conjecture that Borel normality does not imply FS-incompressibility.

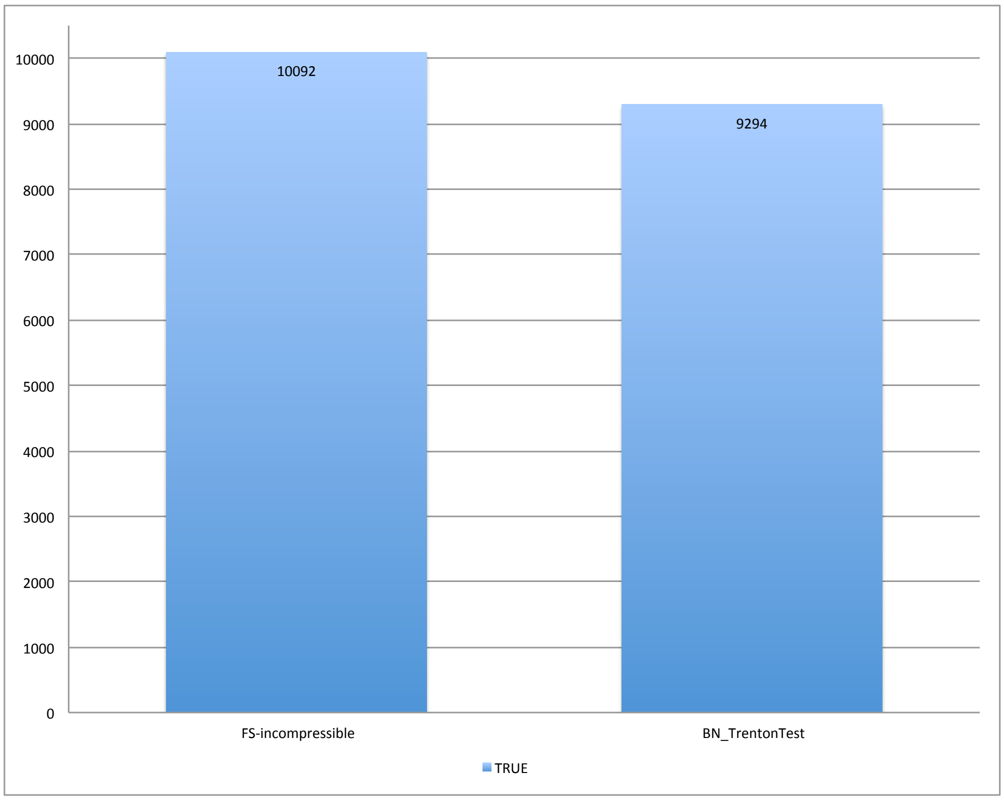

Figure 2.3: Comparing FS-incompressibility and Borel-Normality on 10092 strings of length $\geq 16$ and of FS-complexity $\leq 30$, using $S_1$.

**Conjecture 43.** *For all integers $M$, there exists $x \in \Sigma^*$, $|x| \geq M$ such that $x$ is both Borel normal and FS-compressible.*

We also observe, from our experimental results, that most of the non-Borel normal strings are FS-incompressible (98.38%), but with only 6.60% being of length greater or equal to 33 (40 out of the 606 to be precise). One of these being the string

$$w_3 = 00001000010000100001000010000100001 = (00001)^7.$$

As an aside, note that this counter-example still holds an evident pattern despite being FS-incompressible. We find similar cases in the $S_1$ encoding results, but then all strings are FS-incompressible so far. So, most of our strings are FS-incompressible (all of them for $S_1$), which seems to contradict Conjecture 43. But, the sets that are not Borel normal are very small in both cases, so, in reality, the result is not too surprising. However, what is interesting and supports our conjecture, is that the set of non Borel normal and FS-incompressible strings includes less and less strings as the length of the strings increase. The percentage

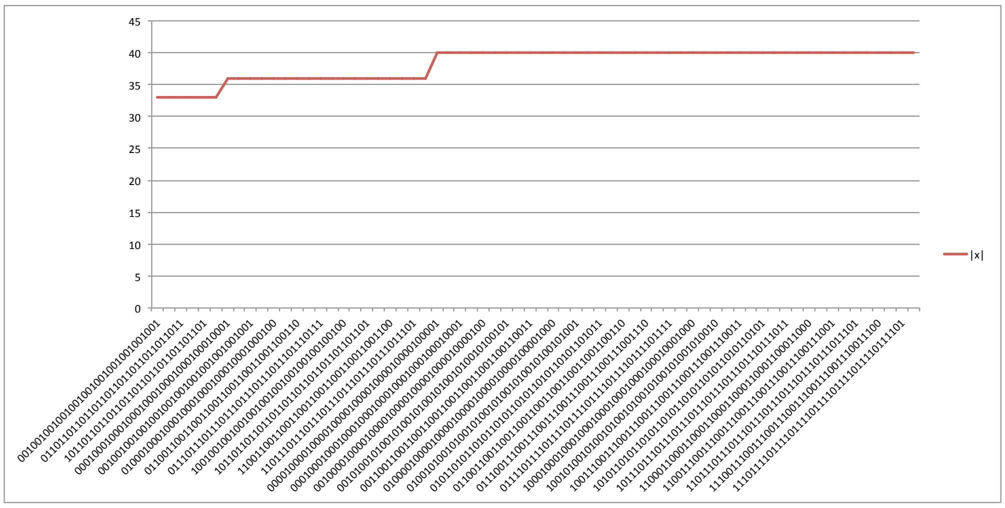

Figure 2.4: Relating the set of Borel normal and FS-compressible strings to their length, using the $S_0$ encoding data.

of these strings decreases really fast, a trend that we can clearly see in both Figures 2.5 and 2.6 below. These show the trends of the strings that are both FS-incompressible and not Borel normal on the two data sets—the $S_0$ and the $S_1$ encoding results.

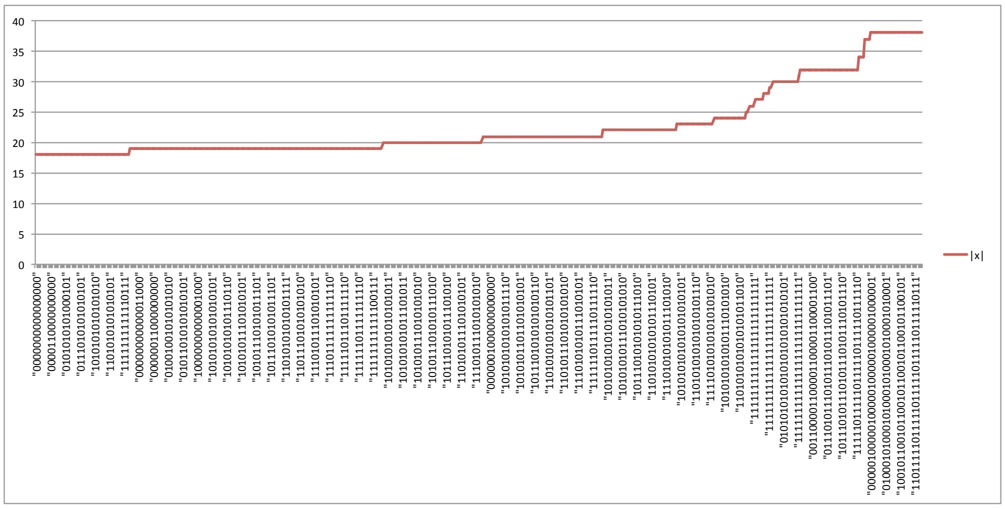

Figure 2.5: Relating the set of non-Borel normal and FS-incompressible strings to their length, using the $S_0$ encoding data. Note that the size of the data set concerned is 606 strings, thus not all appear on the x-axis.

Notice that as we mention above, in the two plots shown in Figures 2.5 and 2.6, the overall trend is that the number of strings concerned drops considerably

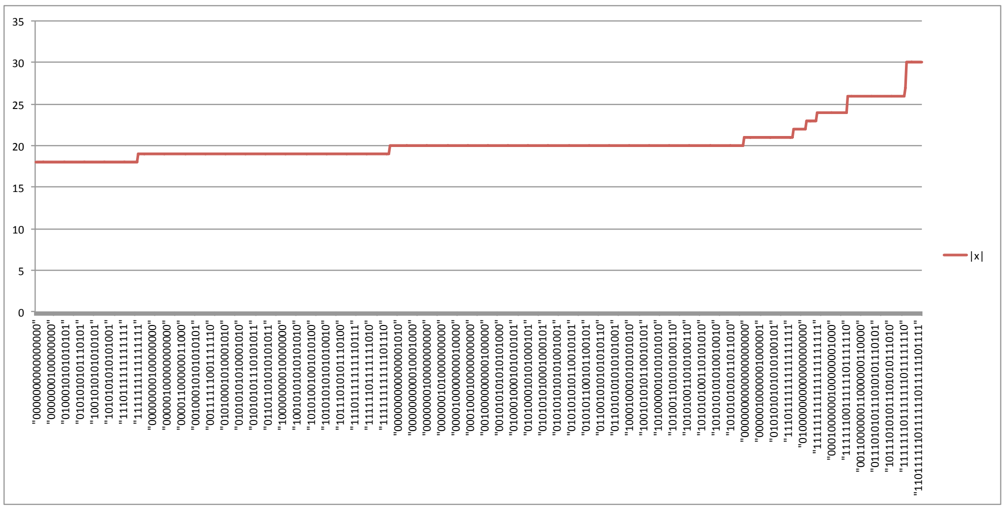

Figure 2.6: Relating the set of Borel normal and FS-compressible strings to their length, using the $S_1$ encoding data. Note that the size of the data set concerned is 798 strings, thus not all appear on the x-axis.

as their length increases. Therefore, we can conjecture that a threshold exists before we can say that all FS-incompressible strings are Borel normal.

**Conjecture 44.** *There exists a constant $M$ such that for all $x \in \Sigma^*$ and $|x| \geq M$, $x$ is finite-state incompressible implies that $x$ is Borel normal.*

Proving Conjectures 43 and 44 would show that for almost all strings Borel normality and finite-state incompressibility are correlated, just as they are in AIT. As for infinite sequences, their Borel normality definition is slightly altered and we are still unaware of the behaviour of finite-state complexity as we tend towards infinity. Hence we cannot draw any conclusions at this point in time. Nevertheless, we suspect that the correlation we mention here is omnipresent in the case of infinite sequences. It is definitely an open question worth studying further.

**Open Problem 45.** *How are finite-state incompressibility and Borel normality correlated in the domain of infinite sequences?*

## 2.3 State-Size Hierarchy

By the *state-size hierarchy* we refer to the hierarchy of languages $L_{\leq m}$, $m \geq 1$, consisting of strings where a minimal description uses a transducer with at most $m$ states. In this section, we show that the state-size hierarchy with respect to the standard encoding is infinite; however, it remains an open question whether this hierarchy is strict at every level.

In a more general setting, the definition of finite-state complexity allows an arbitrary computable encoding of the transducers, and properties of the state-size hierarchy depend significantly on the particular encoding. We establish that, for suitably chosen computable encodings, every level of the state-size hierarchy can be strict.

In order to prove the results related to this section we first need to cover some preliminaries.

### 2.3.1 Preliminaries

By a *computable encoding* of all transducers we mean a pair $S = (D_S, f_S)$ where $D_S \subseteq \Sigma^*$ is a decidable set and $f_S : D_S \to \mathcal{T}_{DGSM}$ is a computable bijective mapping that associates a transducer $T_\sigma^S$ to each $\sigma \in D_S$, where $\mathcal{T}_{DGSM}$ refers to the set of transducers (or deterministic general sequential machines).[4]

We say that $S$ is a *polynomial-time (computable) encoding* if $D_S \in \mathrm{P}$, where P is the class of sets computed in polynomial-time, and for a given $\sigma \in D_S$ we can compute the transducer $T_\sigma^S \in \mathcal{T}_{DGSM}$ in polynomial time. We identify a transducer $T \in \mathcal{T}_{DGSM}$ with its transition function (1.1), and the set of state names is always $\{1, \ldots, |Q|\}$ where 1 is the start state. By "computing the transducer $T_\sigma^S$" we mean an algorithm that (in polynomial time) outputs the list of transitions (corresponding to (1.1), with state names written in binary) of $T_\sigma^S$.

In what follows, we refer to our fixed natural encoding $S_0$ of transducers as the *standard encoding.* For our main result we need some fixed encoding of the transducers where the length of the encoding relates in a 'reasonable way' to the lengths of the transition outputs. Recall that we encode a transducer as a binary string by listing for each state $q$ and input symbol $i \in \Sigma$ the output and target state corresponding to the pair $(q, i)$, that is, $\Delta(q, i)$. Thus, the encoding of a transducer is a list of (encodings of) states and output strings.

The results of Section 2.3.3 remain valid if, as our standard encoding, we would encode the transducers by listing the transitions (that is, the pairs consisting of the target state and the output string encoded in binary) in any reasonable way where an output string $w$ corresponding to an individual transition contributes to the length of the encoding a quantity $c \cdot |w|$, where $c$ is a constant.

In this section we define the encoding $S_0$ as the standard encoding. We denote it as the pair $(D_{S_0}, f_{S_0})$ where $f_{S_0}$ associates to each $\sigma \in D_{S_0}$ the transducer $T_\sigma^{S_0}$ as described above. It can be verified that for each $T \in \mathcal{T}_{DGSM}$ there exists a unique $\sigma \in D_{S_0}$ such that $T = T_\sigma^{S_0}$, that is, $T$ and $T_\sigma^{S_0}$ have the same transition function. The details of the verification procedure to check that $T_{\sigma_1}^{S_0} \neq T_{\sigma_2}^{S_0}$

[4]In a more general setting the mapping $f_S$ may not be injective (for example if we want to define $D_S$ as a regular set [12]). However, in the following we restrict consideration to bijective encodings in order to avoid unnecessary complications with the notation associated with our state-size hierarchy.

when $\sigma_1 \neq \sigma_2$ can be found in [12]. For $T \in \mathcal{T}_{DGSM}$, the *standard encoding of* $T$ is the unique $\sigma \in D_{S_0}$ such that $T = T_\sigma^{S_0}$. The standard encoding $S_0$ is a polynomial-time encoding.

### 2.3.2 Finite-State Complexity and State-Size

Here we are interested in the state-size, that is the number of states of transducers used for minimal encodings of arbitrary strings. For $m \geq 1$ we define the language $L^S_{\leq m}$ to consist of strings $x$ that have a minimal description using a transducer with at most $m$ states. Formally, we write

$$\begin{aligned} L^S_{\leq m} \; = \{ \quad & x \in \Sigma^* \; : \; (\exists \sigma \in D_S, p \in \Sigma^*) \; T^S_\sigma(p) = x, \\ & |\sigma| + |p| = C_S(x), \mathrm{size}(T^S_\sigma) \leq m \}. \end{aligned}$$

By setting $L^S_{\leq 0} = \emptyset$, the set of strings $x$ for which the smallest number of states of a transducer in a minimal description of $x$ is $m$ can then be denoted as

$$L^S_{=m} = L^S_{\leq m} - L^S_{\leq m-1}, \quad m \geq 1.$$

Also, we let $L^S_{\exists_{\min} m}$ denote the set of strings $x$ that have a minimal description in terms of a transducer with exactly $m$ states. Note that $L^S_{=m} \subseteq L^S_{\exists_{\min} m}$, but the converse inclusion need not hold, in general, because strings in $L^S_{\exists_{\min} m}$ may have other minimal descriptions with fewer than $m$ states.

In the following, when dealing with the standard encoding $S_0$ (introduced in Section 2.3.1) we write, for short, $T_\sigma$, $||(T,p)||$, $C$ and $L_{\leq m}$, $L_{=m}$, $L_{\exists_{\min} m}$, $m \geq 1$, instead of $T^{S_0}_\sigma$, $||(T,p)||_{S_0}$, $C_{S_0}$ and $L^{S_0}_{\leq m}$, $L^{S_0}_{=m}$, $L^{S_0}_{\exists_{\min} m}$, respectively. The main result in Section 2.3.3 is proved using the standard encoding; however it could easily be modified for any 'naturally defined' encoding of transducers, where each transducer is described by listing the states and transitions in a uniform way. For example, the more efficient encoding considered in Section 1.3.2 clearly satisfies this property. On the other hand, when dealing with arbitrarily defined computable encodings $S$, the languages $L^S_{\leq m}$, $m \geq 1$, can obviously have very different properties. In the Subsection 2.3.4 we consider properties of the more general computable encodings.

**Proposition 46.** *For any computable encoding $S$, the languages $L^S_{\leq m}$, $m \geq 1$, are decidable.*

We conclude this section with an example concerning the finite-state complexity with respect to the standard encoding.

**Example 47.** Define the sequence of strings

$$w_m = 1010^2 10^3 1 \cdot \ldots \cdot 0^{m-1} 10^m 1, \;\; m \geq 1.$$

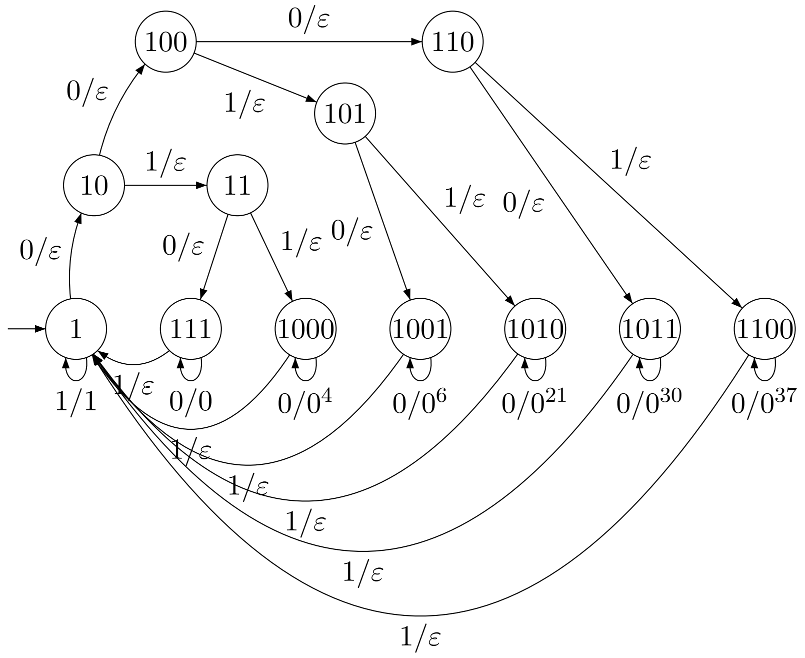

Figure 2.7: The transducer $T_7$ for Example 47.

Using the transducer $T_7$ of Figure 2.7 we produce an encoding of $w_{99}$. Note that $|w_{99}| = 5050$.

With the names of the states indicated in Figure 1, $T_1$ is encoded by a string $\sigma_1 \in S_0$ of length 332. Each number $0 \leq i \leq 99$ can be represented as a sum of, on average, 2.92 numbers from the multi-set $\{1, 4, 6, 21, 30, 37\}$ [37]. Thus, when we represent $w_{99}$ in the form $T_1(p_{99})$, we need on average at most $6 \cdot 2.92$ symbols in $p_{99}$ to output each substring $0^i$, $0 \leq i \leq 99$. (This is only a very rough estimate as it assumes that for each element in the sum representing $i$ we need to make a cycle of length six through the start state, and this is of course not true when the sum representing $i$ has some element occurring more than once.) Additionally we need to produce the 100 symbols '1'. This means that the length of $p_{99}$ can be chosen to be at most 1852. Our estimate gives that

$$||(T_{\sigma_1}, p_{99})|| = |\sigma_1| + |p_{99}| = 2184,$$

which is a very rough upper bound for $C(w_{99})$.

The above estimation could be improved using more detailed information from the computation of the average from [37].

Arguably, these types of constructions are hinting that computing the value of finite-state complexity may have connections to the so-called postage stamp problems considered in number theory, with some variants known to be computationally hard [27, 36]. Consider a set $A_n = a_1, a_2, ..., a_n$ of $n$ positive integer-denomination postage stamps sorted such that $1 = a_1 < a_2 < ... < a_n$. Suppose they are to be used on an envelope with room for no more than $h$ stamps. The postage stamp problem then consists of determining the smallest integer $N_h(A_n)$ which cannot be represented by a linear combination $\sum_{(i=1)}^{n} x_i a_i$ with $x_i \geq 0$ and $\sum_{(i=1)}^{n} x_i < h$. Without the latter restriction this problem is known as the Frobenius problem or Frobenius postage stamp problem. Therefore, if the capacity of the envelope $h$ is fixed it is a polynomial time problem. If the capacity $h$ is arbitrary the problem is known to be NP-hard [36]. We refer you back to Conjecture 23 on the problem NP-hardness concerning finite-state complexity.

### 2.3.3 State-Size Hierarchy

We now establish that finite-state complexity is a rich complexity measure with respect to the number of states of the transducers, in the sense that there is no *a priori* upper bound for the number of states used for minimal descriptions of arbitrary strings. This is in contrast to algorithmic information theory, where the number of states of a universal Turing machine can be fixed.

We prove the hierarchy result using the standard encoding. The particular choice of the encoding is not important and the proof could be easily modified for any encoding that is based on listing the transitions of a transducer in a uniform way. However as we discuss later, arbitrary computable encodings can yield hierarchies with very different properties.

Recall that in this section we use the standard encoding $S_0$ and the specification $S_0$ is dropped as a sub- or superscript in the notations associated with finite-state complexity.

**Theorem 48.** *For any $n \in \mathbb{N}$ there exists a string $x_n$ such that $x_n \notin L_{\leq n}$.*

*Proof.* Consider an arbitrary but fixed $n \in \mathbb{N}$. We define $2n+1$ strings of length $2n+3$,

$$u_i = 10^i 1^{2n+2-i}, \;\; i = 1, \ldots, 2n+1.$$

For $m \geq 1$, we define

$$x_n(m) = u_1^{m^2} u_2^{m^2} \cdots u_{2n+1}^{m^2}.$$

Let $(T_\sigma, p)$ be an arbitrary encoding of $x_n(m)$ where $\text{size}(T_\sigma) \leq n$. We show that by choosing $m$ to be sufficiently large as a function of $n$, we have

$$||(T_\sigma, p)|| > \frac{m^2}{2}. \tag{2.3}$$

The set of transitions of $T_\sigma$ can be written as a disjoint union $\theta_1 \cup \theta_2 \cup \theta_3$, where

- $\theta_1$ consists of the transitions where the output contains a unique $u_i$, $1 \leq i \leq 2n+1$, as a substring, that is, for any $j \neq i$, $u_j$ is not a substring of the output;

- $\theta_2$ consists of the transitions where for distinct $1 \le i < j \le 2n+1$, the output contains both $u_i$ and $u_j$ as a substring;
- $\theta_3$ consists of transitions where the output does not contain any of the $u_i$'s as a substring, $i = 1, \ldots, 2n+1$.

Note that if a transition $\alpha \in \theta_3$ is used in the computation $T_\sigma(p)$, the output produced by $\alpha$ cannot completely overlap any of the occurrences of $u_i$'s, $i = 1, \ldots, 2n+1$. Hence

$$\text{a transition of } \theta_3 \text{ used by } T_\sigma \text{ on } p \text{ has output length at most } 4n+4. \tag{2.4}$$

Since $T_\sigma$ has at most $n$ states, and consequently at most $2n$ transitions, it follows by the pigeon-hole principle that there exists $1 \le k \le 2n+1$ such that $u_k$ is not a substring of any transition of $\theta_1$. We consider how the computation of $T_\sigma$ on $p$ outputs the substring $u_k^{m^2}$ of $x_n(m)$. Let $z_1, \ldots, z_r$ be the minimal sequence of outputs that 'covers' $u_k^{m^2}$. That is, $z_1$ (respectively, $z_r$) is the output of a transition that overlaps with a prefix (respectively, a suffix) of $u_k^{m^2}$ and $u_k^{m^2}$ is a substring of $z_1 \cdots z_r$.

Define

$$\Xi_i = \{1 \le j \le r \mid z_j \text{ is output by a transition of } \theta_i\}, \quad i = 1, 2, 3.$$

By the choice of $k$ we know that $\Xi_1 = \emptyset$. For $j \in \Xi_2$, we know that the transition outputting $z_j$ can be applied only once in the computation of $T_\sigma$ on $p$ because for $i < j$ all occurrences of $u_i$ as substrings of $x_n(m)$ occur before all occurrences of $u_j$. Thus, for $j \in \Xi_2$, the use of this transition contributes at least $2 \cdot |z_j|$ to the length of the encoding $||(T_\sigma, p)||$.

Finally, by (2.4), for any $j \in \Xi_3$ we have $|z_j| \le 4n+4 < 2|u_k|$. Such transitions may naturally be applied multiple times, however, the use of each transition outputting $z_j$, $j \in \Xi_3$, contributes at least one symbol to $p$.

Thus we get the following estimate:

$$||(T_\sigma, p)|| \ge \sum_{j \in \Xi_2} 2 \cdot |z_j| + |\Xi_3| > \frac{|u_k^{m^2}|}{2|u_k|} = \frac{m^2}{2}.$$

To complete the proof it is sufficient to show that, with a suitable choice of $m$, $C(x_n(m)) < \frac{m^2}{2}$. The string $x_n(m)$ can be represented by the pair $(T_1, p_1)$ where $T_1$ is the $2n$-state transducer from Figure 2.8 and $p_1 = (0^m1)^{2n-1}0^m1^m$.

Each state of $T_1$ can be encoded by a string of length at most $\lceil \log_2(2n) \rceil$. We recall that in the standard encoding each transition output $v$ contributes $|v^\diamond| = 2|v| + 2$ to the length of the encoding, and each binary encoding $u$ of a state name that is the target of a transition and that is not a self-loop contributes

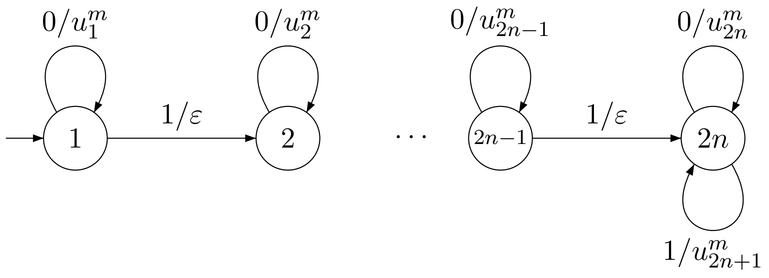

Figure 2.8: The transducer $T_1$ from the proof of Theorem 48.

$2|u|$ to the length of the encoding. So, we get the following upper bound for the length of a string $\sigma_1 \in S_0$ encoding $T_1$:

$$|\sigma_1| \leq (8n^2 + 16n + 8)m + (4n - 2)(\lceil \log_2(2n) \rceil + 1).$$

Noting that $|p_1| = (2n+1)m + 2n - 1$ we observe that

$$C(x_n(m)) \leq ||(T_{\sigma_1}, p_1)|| = |\sigma_1| + |p_1| < \frac{m^2}{2}. \tag{2.5}$$

For example if we choose $m = 16n^2 + 36n + 19$. This completes the proof. □

As a corollary we obtain that the sets of strings with minimal descriptions using a transducer with at most $m$ states, $m \geq 1$, form an infinite hierarchy.

**Corollary 49.** *For any $n \geq 1$, there exists effectively $k_n \geq 1$ such that $L_{\leq n} \subset L_{\leq n+k_n}$.*[5]

We do not know whether all levels of the state-size hierarchy with respect to the standard encoding are strict. Note that the proof of Theorem 48 constructs strings $x_n(m)$ that have a smaller description using a transducer with $2n$ states than any description using a transducer with $n$ states. We believe that (with $m$ chosen as in the proof of Theorem 48) the minimal description of $x_n(m)$ has in fact $2n$ states, but do not have a complete proof for this claim. The claim would imply that $L_{\leq n}$ is strictly included in $L_{\leq 2n}$, $n \geq 1$. In any case the construction used in the proof of Theorem 48 gives an effective upper bound for the size of $k_n$ such that $L_{\leq n} \subset L_{\leq n+k_n}$, because the estimation (2.5) (with the particular choice for $m$) implies also an upper bound for the number of states of a transducer used in a minimal description of $x_n(m)$.

The standard encoding is monotonic in the sense that adding states to a transducer or increasing the lengths of the outputs, always increases the length of an encoding. This leads us to believe that for any $n$ there exist strings where the minimal transducer has exactly $n$ states, that is, for any $n \geq 1$, $L_{=n} \neq \emptyset$.

[5]Note that here "$\subset$" stands for strict inclusion.

**Conjecture 50.** *$L_{\leq n} \subset L_{\leq n+1}$, for all $n \geq 1$.*

By Proposition 46 we know that the languages $L_{\leq n}$ are decidable. Thus for $n \geq 1$ such that $L_{=n} \neq \emptyset$, in principle, it would be possible to compute the length $\ell_n$ of shortest words in $L_{=n}$. However we do not know how $\ell_n$ behaves as a function of $n$. Using a brute-force search we have established [12] that all strings of length at most 23 have a minimal description using a single state transducer.

**Open Problem 51.** *What is the asymptotic behavior of the length of the shortest words in $L_{=n}$ as a function of $n$?*

In addition, we do not know whether there exists $x \in \Sigma^*$ that has two minimal descriptions (in the standard encoding) where the respective transducers have different numbers of states. This amounts to the following.

**Open Problem 52.** *Does there exist an $n \geq 1$ such that $L_{=n} \neq L_{\exists_{min} n}$?*

### 2.3.4 General Computable Encodings

While the proof of Theorem 48 can be easily modified for any encoding that, roughly speaking, is based on listing the transitions of a transducer, the proof breaks down if we consider arbitrary computable encodings $S$. Note that the number of finite transducers (with $n$ states) is infinite and, for arbitrary computable $S$, it does not seem easy, analogously to the proof of Theorem 48, to get upper and lower bounds for $C_S(x_n(m))$ for suitably chosen strings $x_n(m)$. We do not know whether there exist computable encodings for which the state-size hierarchy collapses to a finite number of levels.

**Open Problem 53.** *Does there exist an $n \geq 1$ and a computable encoding $S_n$ such that that, for all $k \geq 1$, $L^{S_n}_{\leq n} = L^{S_n}_{\leq n+k}$?*

On the other hand, it is possible to construct particular encodings for which every level of the state-size hierarchy is strict.

**Theorem 54.** *There exists a computable encoding $S_1$ such that*

$$L^{S_1}_{\leq n-1} \subset L^{S_1}_{\leq n}, \quad \textit{for each } n \geq 1.$$

*Proof.* Let $p_i$, $i = 1, 2, \ldots$, be the $i$th prime. We define an $n$-state ($n \geq 1$) transducer $T_n = (\{1, \ldots, n\}, 1, \Delta_n)$ by setting $\Delta_n(1, 0) = (1, 0^{p_n})$, $\Delta_n(i, 0) = (i, \varepsilon)$, $2 \leq i \leq n$, $\Delta_n(j, 1) = (j + 1, \varepsilon)$, $1 \leq j \leq n - 1$, and $\Delta_n(n, 1) = (n, \varepsilon)$.

In the encoding $S_1$ we use the string $\sigma_n = \text{bin}\,(n)$ to encode the transducer $T_n$, $n \geq 1$. Any transducer $T$ that is not one of the above transducers $T_n$, $n \geq 1$, is encoded in $S_1$ by a string $0 \cdot e$, $e \in \Sigma^*$, where $|e|$ is at least the sum of the lengths of outputs of all transitions in $T$. This condition is satisfied, for example by choosing the encoding of $T$ in $S_1$ to be simply 0 concatenated with the standard encoding of $T$.

Let $m \geq 1$ be arbitrary but fixed. The string $0^{p_m}$ has a description $(T^{S_1}_{\sigma_m}, 0)$ of size $\lceil \log_2 m \rceil + 1$, where $\sigma_m \in S_1$ encodes $T_m$ and the transducer $T^{S_1}_{\sigma_m}$ has $m$ states. We show that $C_{S_1}(0^{p_m}) = \lceil \log_2 m \rceil + 1$.

By the definition of the transducers $T_n$, for any $w \in \Sigma^*$, $T_n(w)$ is of the form $0^{k \cdot p_n}$, $k \geq 0$. Thus, $0^{p_m}$ cannot be the output of any transducer $T_n$, $n \neq m$.

On the other hand, consider an arbitrary description $(T^{S_1}_{\sigma}, w)$ of the string $0^{p_m}$ where $T^{S_1}_{\sigma}$ is not any of the transducers $T_n$, $n \geq 1$. Let $x$ be the length of the longest output of a transition of $T^{S_1}_{\sigma}$. Thus, $x \cdot |w| \geq p_m$. By the definition of $S_1$ we know that $|\sigma| \geq x + 1$ and we conclude that

$$||(T^{S_1}_{\sigma}, w)||_{S_1} = |\sigma| + |w| > \lceil \log_2 m \rceil + 1.$$

In the encoding $S_1$ we have shown that the unique minimal description of $0^{p_m}$ uses a transducer with $m$ states, which implies $0^{p_m} \in L^{S_1}_{=m}$, $m \geq 1$. □

The encoding $S_1$ constructed in the proof of Theorem 54 is not a polynomial-time encoding because $T_n$ has an encoding of length $O(\log_2 n)$, whereas the description of the transition function of $T_n$ (in the format specified in Subsection 1.3.1) has length $\Omega(n \cdot \log_2 n)$. Besides the above problem $S_1$ is otherwise efficiently computable and using standard 'padding techniques' we can simply increase the length of all encodings of transducers in $S_1$.

**Corollary 55.** *There exists a polynomial time encoding $S'_1$ such that*

$$L^{S'_1}_{\leq n-1} \subset L^{S'_1}_{\leq n}, \quad \textit{for each } n \geq 1.$$

*Proof.* The encoding $S'_1$ is obtained by modifying the encoding $S_1$ of the proof of Theorem 54 as follows. For $n \geq 1$, $T_n$ is encoded by the string $\sigma_n = \text{bin}^\dagger(n) \cdot 1^n$. Any transducer $T$ that is not one of the transducers $T_n$, $n \geq 1$, is encoded by a string $0 \cdot w$ where $|w| \geq 2^x$ and $x$ is the sum of the lengths of outputs of all transitions of $T$. If $\sigma$ is the standard encoding of $T$, then, we can choose $w = \sigma^\dagger \cdot 1^{2^{|\sigma|}}$, for example.

Now $|\sigma_n|$ is polynomially related to the length of the description of the transition function of $T_n$, $n \geq 1$, and given $\sigma_n$ the transition function of $T_n$ can be output in quadratic time. For transducers not of the form $T_n$, $n \geq 1$, the same holds trivially.

Essentially in the same way as in the proof of Theorem 54, we verify that for any $m \geq 1$, the string $0^{p_m}$ has a unique minimal description $(T^{S'_1}_{\sigma'_m}, 0)$, where $\sigma'_m \in S'_1$ is the description of the $m$-state transducer $T_m$. The same argument works because the encoding of any transducer $T$ in $S'_1$ is roughly speaking obtained from the encoding $\sigma$ of $T$ in $S_1$ by appending $2^{|\sigma|}$ '1' symbols. □

There exist computable encodings that allow distinct minimal descriptions of strings based on transducers with different numbers of states. Furthermore, the gap between the numbers of states of the transducers used for different minimal descriptions of the same string can be made arbitrarily large. That is, for any $n < m$ we can construct an encoding where some string has minimal descriptions both using transducers with either $n$ or $m$ states. The proof uses an idea similar to the proof of Theorem 54.

**Theorem 56.** *For any $1 \leq n < m$, there exists a computable encoding $S_{n,m}$ such that $L^{S_{n,m}}_{\exists_{min} m} \cap L^{S_{n,m}}_{=n} \neq \emptyset$.*

*Proof.* Let $p_i$, $i = 1, 2, \ldots$, be the $i$th prime. Let $T_i$, $i \geq 1$, be the particular transducers defined in the proof of Theorem 54 and let $S_1$ be the encoding defined there.

Let $1 \leq n < m$ be arbitrary but fixed. We denote by $T'_n = (\{1, \ldots, m\}, 1, \Delta')$ an $m$-state transducer where $\Delta'(1, 0) = (1, 0^{p_n})$, $\Delta'(i, 0) = (1, \varepsilon)$, $2 \leq i \leq m$, $\Delta'(j, 1) = (j + 1, \varepsilon)$, $1 \leq j \leq m - 1$, and $\Delta'(m, 1) = (m, \varepsilon)$. The transducer $T'_n$ is obtained from $T_n$ simply by adding $m - n$ 'useless' states.

Let $S_2$ be defined as $S_1$ except that the transducer $T_i$, $i \geq 1$, is encoded by the string $\sigma_i = \text{bin}\,(i) \cdot 0$. In the encoding $S_2$ the unique minimal description of the string $0^{p_i}$ is $(T^{S_2}_{\sigma_i}, 0)$. Note that the encoding of $T_i$ in $S_2$ has one additional bit compared to the encoding of $T_i$ in $S_1$. However the same estimation as used in the proof of Theorem 54 goes through.

Now we can choose $S_{n,m}$ to be as $S_2$ except that the transducer $T'_n$ is encoded as $\tau_n = \text{bin}\,(n) \cdot 1$. In the encoding $S_{n,m}$ the minimal description of $0^{p_n}$ using a transducer with the smallest number of states is $(T^{S_{n,m}}_{\sigma_n}, 0)$. However $0^{p_n}$ has another minimal description $(T^{S_{n,m}}_{\tau_n}, 0)$ where the transducer has $m$ states. $\square$

Note that the statement of Theorem 56 implies that $L^{S_{n,m}}_{=m} \neq L^{S_{n,m}}_{\exists_{\min} m}$. Again, by padding the encodings as in Corollary 55, the result of Theorem 56 can be established using a polynomial-time encoding.

## 2.4 Summary

In this chapter, we have reintroduced and developed the first steps of a variant of AIT based on finite transducers [12, 13]. The finite-state complexity, central to the new theory, is computable and satisfies a strong form of the Invariance Theorem.

As this is a computable complexity, in Section 2.2 we explored ways to develop an effective and efficient algorithm to compute it and to start building the infinite table of finite-state complexity measures. We also proffered the initial analysis of the sets of complexities from this algorithm and infered some preliminary results about this new theory, especially concerning its relation to Borel normality.

In contrast to descriptional complexities from AIT, there is no *a priori* upper bound for the number of states used for minimal descriptions of arbitrary strings, which brought our attention to the matter of state-size [13, 14]. As perhaps expected, the properties of the state-size hierarchy with respect to the specific computable encodings considered in Subsection 2.3.4 could be established using constructions where we added to transducers additional states without changing the size of the encoding. In a similar way, various other properties can be established for the state-size hierarchy corresponding to specific (artificially defined) computable encodings. The main open problem concerning general computable encodings is whether or not it is possible to construct an encoding for which the state-size hierarchy collapses to some finite level (see Open problem 53).

As our main result in Section 2.3, we have established that the state-size hierarchy with respect to the standard encoding is infinite. An almost identical proof can be used to show that the hierarchy is infinite with respect to any 'natural' encoding that is based on listing the transitions of the transducer in some uniform way. Many interesting open problems dealing with the hierarchy with respect to the standard encoding remain. In addition to the problems discussed in Subsection 2.3.3, we can consider various types of questions related to combinatorics on words. For example, assuming that a minimal description of a string $w$ needs a transducer with at least $m$ states, is it possible that $w^2$ has a minimal description based on a transducer with less than $m$ states? We conjecture a negative answer to this question.

**Conjecture 57.** *If $w \in L_{=m}$ $(m \geq 1)$, then for any $k \geq 1$, $w^k \notin L_{\leq m-1}$.*

Chapter 3

# Towards a Practical Application: DNA Compression

In this chapter we present the beginnings of an application of finite-state complexity, with which we aim to measure the finite-state complexity of some deoxyribo nucleic acid (DNA) sequences. These results could then be used for DNA compression amongst other further practical applications. DNA compression is not a new concept in the field of bioinformatics, and many algorithms have already been devised for that purpose. In fact, Rivals *et al.* [32] offer the first text compression scheme targeted at DNA sequences: *Cfact*. Their successful scheme is also proven to guarantee compression for repetitive DNA sequences and out-performs most state-of-the-art text compressors [32]. Other DNA-dedicated compressors and pattern recognisors have also been developed which are mostly grammar-based (SEQUITUR, LONGEST MATCH, RE-PAIR, etc.) [15, 19]. We will further discuss the concept of grammar-based compression shortly.

Our original interest was to use the existing results and algorithms to measure the finite-state complexity of some smaller DNA sequences such as *transposons*. A transposon is a segment of DNA that can become integrated at many different sites along a chromosome (especially a segment of bacterial DNA that can be translocated as a whole). Since DNA uses a different alphabet—$\{A, C, G, T\}$ instead of the $\Sigma$ we have been using throughout this thesis—we need to represent or translate that alphabet into a uniquely decodable binary alphabet: $\{00, 01, 10, 11\}$; in order to measure the complexity of DNA. This way we are manipulating and measuring a more specific family of binary strings. However, the algorithms computing the finite-state complexity of a string are limited to a brute-force approach. Therefore they are restricted to an exponential growth depending on the size of the string we wish to measure. Moreover, our current table is not complete enough to use it directly in this case, and the sequences are too large to hope for an exact measurement. Both these obstacles mean that the original goal was too ambitious.

Nevertheless, Carrascosa *et al.* [15] offer both optimisations of the existing algorithms and a new state-of-the-art algorithm to solve the smallest grammar problem: Iterative Repeat Replacement Minimal Grammar Parsing (IRRMGP). This is a problem often related to DNA compression. The smallest grammar problem is the problem of finding the smallest context-free grammar which encodes for a unique string of characters; conventionally the class of straight-line grammars is used to attempt to solve this problem. IRRMGP provides a smallest context-free grammar (in particular, a straight-line grammar) which generates exactly one string—the string of interest. Hence, we have worked on approximating the complexity by attempting to lower the upper bound with Dr. F. Coste [21].

Thus in this chapter we present our general approach: to merge the concept of solving the smallest grammar problem and converting it into an equivalent minimal transducer (with its corresponding input, or control sequence) in order to generate the string of interest. The two approaches presented here are a top-down and bottom-up approach. The 'direction' of the approach depends on the ordering of the process to convert the grammar. In Section 3.2, we then discuss the development of the score function used in an updated version of IRRMGP to guide the SLG construction to our advantage.

## 3.1 Conversion Algorithms for Transforming a Smallest Grammar into a Transducer

In this section we present the algorithms devised to convert a SLG into an equivalent transducer. Before we begin to introduce the conversion algorithms, we need to cover some preliminaries. We especially need to define our concept of 'equivalence' between a straight-line grammar and a transducer, which is a term we use more generously than is conventionally encountered.

### 3.1.1 Preliminaries

Here we state that a transducer is equivalent to a straight-line grammar when the said transducer simulates the grammar in its possible runs. That is, when the language of the grammar is a subset of the language of the transducer, denoted

$$L(G) \subseteq L(T).$$

We do need to note that the language of a SLG, $L(G)$, is in fact the singleton $\{x\}$ since a SLG outputs a unique string $x$ (see Section 1.2). Therefore, all we need in our (generous) definition of equivalence between a SLG and a transducer is that $x \in L(T)$.

Obviously there are many equivalent transducers for one grammar (and vice versa). Here we are interested in the smallest possible equivalent transducer that we can build from a given straight-line grammar. The Subsections 3.1.2 and 3.1.3 offer two methods of doing this.

The algorithms we present follow two versions of the same basic 'guideline'. This guideline is the ordering of the rules of the grammar, which are processed one by one to build the equivalent transducer. Once the transducer is built, a backtracking routine is executed to 'recover' the input string corresponding to our string of interest from the transducer. The computation completes by returning the constructed pair. What follows is a more technical overview of the basic algorithm and of its two variations.

The guideline we referred to is called the *hierarchy ordering*, as it imposes a hierarchy of those rules. The first version is defined as a *top-down ordering* of the grammar rules. This ordering sorts the rules in an 'inter-call' order, which is an increasing ordering of the rules according to how many times each one is called. The way in which a rule is called in a grammar is by having their associated variable (the *lhs* of the rule) appear in the *rhs* of the set rules. As we are working with SLGs, we can safely assume that no rule calls itself and that the start rule is never called (recall that SLGs are context-free grammars). Clearly, this top-down ordering is a partial-order since two rules may be called the same amount of times without ever calling each other, therefore not imposing an order between them. Moreover, the top-down ordering will always list the start rule first. The second version of the hierarchy order is the natural counterpart of top-down ordering: *bottom-up ordering.* This ordering sorts all of the rules in a decreasing order of how many times they are called. For the same reasons as we noted with the former ordering, this is a partial-ordering of the rules and the start rule is never called. It will therefore be the last listed rule.

Once the rule ordering is completed, the algorithms process each rule individually converting these rules into a corresponding subpart of the transducer: two states (or nodes) and a linking transduction (or transition) between them. The transduction holds the label '**null**/$\eta_i$' for the rule $r_i$ (see Chapter 1 for the notation). **null** is a placeholder for a corresponding alphabet symbol, which is only allocated after the entire grammar is processed and the transducer is made 'legal'. By legal we mean that it is deterministic and complete: allowing exactly two out-transductions per state—if we visualise the transducer as a graph, it would be forcing each node to have out-degree 2. After the 'sub-transducer' is built, the algorithm incorporates it into the transducer, and deletes all occurrences of the rule that is being processed from the existing transducer. In order to do so, every transition label is searched for a call of the rule. For every call the variable is deleted from that transition and a path to the new 'sub-transducer' is added.

Because each rule (that is not the start rule) is called at least twice, the subpart of the transducer representing this rule will have nodes with in- and out-degrees strictly greater than 2. This is where the process of making the transducer deterministic is necessary. However, the deterministic process is not put into motion until after the entire grammar has been converted. After processing each rule, the algorithm minimises the transducer. Finally, once all the rules have been processed, we have a completed legal transducer and the input corresponding to the string of interest—the one obtained by the SLG—is obtained using a backtracking algorithm.

All the algorithms in this section are presented in their most general form (except where indicated), so that they can be applied to as wide a range of parameters as we could envision: different sized alphabets, different encodings, different smallest grammar inputs, etc. The methods which are either in reference to well-known algorithms or trivial/obvious procedures are not detailed in the following algorithms. For the sake of simplicity in the pseudocode, we also assume that the transducer $T$ is a global variable and that when $T$'s $\Delta$ function needs to be 'traversed' it is done so in a breadth-first search (BFS) manner from the initial state.

The implementations were conceived to be written Java. Hence, we make use of the OOP to implement the straight-line grammar and the transducer as objects. Of course these are only one possible implementation approach. There are undoubtedly possible optimisations which we did not have time to explore for the scope of this work.

This section offers the two theoretically finalised algorithms devised to convert a smallest grammar (in a straight-line form) into an equivalent transducer.

### 3.1.2 Top-Down Approach

In this algorithm we order the rules of the straight-line grammar in the top-down partial-order to guarantee we have a legal grammar ordering for our algorithm. This ordering allows a simplification of the rule processing when we move onto the conversion of the grammar into a transducer. The rule processing is why the top-down approach is crucial. It guarantees that no variable (non-terminal) is repeated in the transducer and that we have a minimal amount of states, based on the given grammar. We require the first rule of the grammar to be the initial/start rule (associated with the start variable); however, as discussed in Section 3.1.1, this occurs naturally in the ordering procedure. Finally, once the transducer is built and completed, we obtain the corresponding input (or control sequence) such that the resulting pair $(T, control)$ outputs $x$ (i.e. $T(control) = x$).

Throughout the pseudocode presented below we give brief descriptions in form of comments before each procedure. After the pseudocode is given in full we give a simple example of its application on an arbitrary string.

**Algorithm 3.1.1:** TOPDOWNSLGTOTRANSDUCERWITHMERGE$(G, x)$

```
 main
  $G \leftarrow$ HIERARCHICALORDER$(G)$
  $q_0 \leftarrow$ CREATENEWSTATE$()$
  $q_f \leftarrow$ CREATENEWSTATE$()$
  $t \leftarrow (q_0, \textbf{null}, q_f, \text{LHS}(r_0))$
  ADD$(\Delta, t)$
  $T \leftarrow (\{q_0, q_f\}, \Sigma, q_0, \Delta)$
  for each $r \in R$
    do RULEPROCESSING$(r)$
  MAKEDETERMINISTIC$()$
  $control \leftarrow$ FINDCONTROLSEQ$(x)$
  return $(T, control)$
```

**Brief:** The following procedure manages the ordering of the rules of the grammar according to their appearance in the *rhs* of the rules by pre-processing the grammar $G$ to initialise the required objects, calling the PARTIALORDER() procedure on those objects and rearranging the grammar according to the partial order. Note that we assume $r_0$ (first encountered rule in $R$) to be the rule associated to $S$ as we are using the top-down ordering. Once the process is complete it returns the rearranged grammar.

```
procedure HIERARCHICALORDER$(G)$
 for each $r \in R$
   do {
     for each $N \in V$
       do if ISSUBSTRING$(\text{RHS}(r), N)$
       then ADD$(children, N)$
     $node \leftarrow (\text{LHS}(r), children)$
     ADD$(nodeArray, node)$
     $n = 0$
     ADD$(partialOrdering, (\text{LHS}(r), n))$
   }
 $partialOrdering \leftarrow$ PARTIALORDER$(partialOrdering, nodeArray, \text{LHS}(r_0), 0)$
 $G \leftarrow$ MAP$(G, partialOrder)$
 return $(G)$
```

**Brief:** This procedure recursively builds the partial ordering of the rules according to the top-down ordering and returns the resulting partial order.

```
procedure PARTIALORDER($partialOrdering, nodeArray, node, orderValue$)
 $n \leftarrow$ GETPARTIALORDERVALUE($partialOrdering, node$)
 $n \leftarrow$ MAX($n, orderValue$)
 $partialOrdering \leftarrow$ SETPARTIALORDERVALUE($partialOrdering, node, n$)
 $children \leftarrow$ GETCHILDREN($nodeArray, node$)
 $m \leftarrow n + 1$
 for each $c \in children$
   do $partialOrdering \leftarrow$ PARTIALORDER($partialOrdering, nodeArray, c, m$)
 return ($partialOrdering$)
```

**Brief:** Processes each given rule $r$ by finding each instance of $N$ (the $lhs$ of $r$) in $T$ and replacing it with $\eta$ (the $rhs$ of $r$), without repetitions. Finally, it induces state minimisation.

```
procedure RULEPROCESSING($r$)
 $N \leftarrow$ LHS($r$)
 $\eta \leftarrow$ RHS($r$)
 $t \leftarrow$ FINDANDISOLATEFIRSTOCCURRENCEINT($N, \eta$)
 $q_s \leftarrow$ GETSOURCESTATE($t$)
 $q_t \leftarrow$ GETTARGETSTATE($t$)
 FINDANDREPLACEALLOCCURRENCESINT($N, q_s, q_t$)
 MINIMISESTATES()
```

**Brief:** Searches for the first occurrence of the non-terminal $N$ in $T$, replaces it with a new transition labelled $\eta$ and returns that transition. It returns **null** if no such transition is found.

```
procedure findAndIsolateFirstOccurrenceInT($N, \eta$)
 for each $t \in \Delta$
   do
     $label \leftarrow$ getLabel($t$)
     if isSubstring($label, N$)
       then
         $pos \leftarrow$ findSubstring($label, N, 0$)
         $end \leftarrow pos + |N|$
         if $pos = 0$ and $end = |label|$
           then
             $t \leftarrow$ (getSourceState($t$), null, getTargetState($t$), $\eta$)
           else if $pos = 0$ and $end \neq |label|$
           then
             $q \leftarrow$ createNewState()
             $t \leftarrow$ (getSourceState($t$), null, $q, \eta$)
             $target \leftarrow$ getTargetState($t$)
             $out \leftarrow$ getSubstring($label, end, |label|$)
             $e \leftarrow (q$, null, $target, out)$
             add($\Delta, e$)
           else if $pos \neq 0$ and $end = |label|$
           then
             $q \leftarrow$ createNewState()
             $t \leftarrow (q$, null, getTargetState($t$), $\eta$)
             $source \leftarrow$ getSourceState($t$)
             $out \leftarrow$ getSubstring($label, 0, pos$)
             $e \leftarrow (source$, null, $q, out)$
             add($\Delta, e$)
           else if $pos \neq 0$ and $end \neq |label|$
           then
             $q_s \leftarrow$ createNewState()
             $q_t \leftarrow$ createNewState()
             $t \leftarrow (q_s$, null, $q_t, \eta)$
             $source \leftarrow$ getSourceState($t$)
             $target \leftarrow$ getTargetState($t$)
             $out1 \leftarrow$ getSubstring($label, 0, pos$)
             $out2 \leftarrow$ getSubstring($label, end, |label|$)
             $e \leftarrow (source$, null, $q_s, out1)$
             $f \leftarrow (q_t$, null, $target, out2)$
             add($\Delta, f$)
             add($\Delta, e$)
         return ($t$)
 return (null)
```

**Brief:** The following searches for every occurrence of $N$ in $T$, replaces it by linking in the existing corresponding transition, whose source and target states are $q_s$ and $q_t$ respectively.

```
procedure FINDANDREPLACEALLOCCURRENCESINT($N, q_s, q_t$)
 for each $e \in \Delta$
  do
   while ISSUBSTRING(GETLABEL($e$), $N$)
    then
     $label \leftarrow$ GETLABEL($e$)
     $pos \leftarrow$ FINDSUBSTRING($label, N, 0$)
     $end \leftarrow pos + |N|$
     if $pos = 0$ and $end = |label|$
       then
         MERGE($q_s$, GETSOURCESTATE($e$))
         MERGE($q_t$, GETTARGETSTATE($e$))
       else if $pos = 0$ and $end \neq |label|$
       then
         MERGE($q_s$, GETSOURCESTATE($e$))
         $target \leftarrow$ GETTARGETSTATE($e$)
         $out \leftarrow$ GETSUBSTRING($label, end, |label|$)
         $f \leftarrow (q_t, \textbf{null}, target, out)$
         ADD($\Delta, f$)
       else if $pos \neq 0$ and $end = |label|$
       then
         MERGE($q_t$, GETTARGETSTATE($e$))
         $source \leftarrow$ GETSOURCESTATE($e$)
         $out \leftarrow$ GETSUBSTRING($label, 0, pos$)
         $f \leftarrow (source, \textbf{null}, q_s, out)$
         ADD($\Delta, f$)
       else if $pos \neq 0$ and $end \neq |label|$
       then
         $source \leftarrow$ GETSOURCESTATE($e$)
         $target \leftarrow$ GETTARGETSTATE($e$)
         $out1 \leftarrow$ GETSUBSTRING($label, 0, pos$)
         $out2 \leftarrow$ GETSUBSTRING($label, end, |label|$)
         $f \leftarrow (source, \textbf{null}, q_s, out1)$
         $g \leftarrow (q_t, \textbf{null}, target, out2)$
         ADD($\Delta, f$)
         ADD($\Delta, g$)
     REMOVE($\Delta, e$)
```

**Brief:** Minimises the number of states by searching for unreachable and dead-end states and deleting them accordingly.

```
procedure minimiseStates()
 for each $q \in Q$
   do if inDegree($q$) $= 0$
        then for each $e \in \Delta$
               do if getSourceState($e$) $= q$
                    then Remove($\Delta, e$)
             Remove($Q, q$)
 for each $q \in Q$
   do if outDegree($q$) $= 0$
        then for each $e \in \Delta$
               do if getTargetState($e$) $= q$
                    then $s \leftarrow$ getSourceState($e$)
                         $in \leftarrow$ getInputSymbol($e$)
                         $out \leftarrow$ getLabel($e$)
                         $f \leftarrow (s, in, s, out)$
                         add($\Delta, f$)
                         Remove($\Delta, e$)
             Remove($Q, q$)
```

**Brief:** This procedure calls for an out-degree fix, via fixOutDegree(), and 'fills in' each transition input symbol 'slot' (currently **null**) with a corresponding alphabet symbol, making the transducer deterministic and complete.

```
procedure makeDeterministic()
 fixOutDegree()
 for each $q \in Q$
   do $a \in \Sigma$
      for each $t$ such that getSourceState($t$) $= q$
        do remove($\Delta, t$)
           $target \leftarrow$ getTargetState($t$)
           $t \leftarrow (target, a, target,$ getLabel($t$))
           add($\Delta, t$)
           $a \leftarrow$ nextSymbol($\Sigma$)
```

**Brief:** Guarantees that the out-degree of every state is $|\Sigma|$ by traversing the set of states $Q$ of the transducer $T$ using a BFS. We leave the BFS for the implementation level as it would overcomplicate the pseudocode of this simple procedure.

```
procedure fixOutDegree()
 for each $q \in Q$
   do
     while $(\textsc{outDegree}(q) - 2) > 0$
       do
         $p \leftarrow \textsc{createNewState}()$
         $count = 0$
         $transitionPair \leftarrow$ null
         for each $t$ such that $\textsc{getSourceState}(t) = q$
           do
             if $count < 2$
               then
                 $\textsc{add}(transitionPair, t)$
                 $count \leftarrow count + 1$
               else break
         for each $t \in transitionPair$
           do
             $e \leftarrow (p, \textbf{null}, \textsc{getTargetState}(t), \textsc{getLabel}(t))$
             $\textsc{add}(\Delta, e)$
             $\textsc{remove}(\Delta, t)$
         $s \leftarrow (q, \textbf{null}, p, \varepsilon)$
         $\textsc{add}(\Delta, s)$
     while $\textsc{outDegree}(q) < 2$
       do
         $e \leftarrow (q, \textbf{null}, q, \varepsilon)$
         $\textsc{add}(\Delta, e)$
```

**Brief:** Traverses $T$ and, in a reverse engineering sort of way, obtains the corresponding *control* sequence to $x$, such that $T(control) = x$.

```
procedure findControlSeq($x$)
 $index \leftarrow 0$
 for each $q \in Q$
   do
     for each $t$ such that $\textsc{getSourceState}(t) = q$
       do
         $label \leftarrow \textsc{getLabel}(t)$
         if $(index + |label|) < |x|$
           then
             if $\textsc{isSubstring}(x, label, index)$
               then
                 $control \leftarrow control \cdot \textsc{getInputSymbol}(t)$
                 $index \leftarrow index + |label|$
 return $(control)$
```

Now we give an applied example of the algorithm in order to illustrate its workings.

**Example 58.** Let $x = 01010100011010110100111101$. Note that $x$ is an arbitrary string and has no relation to DNA sequences. This is just for demonstration purposes.
IRRMGP gives the following SLG for $x$:

$$\begin{aligned}
0 &\rightarrow \backslash 0\ 12\ \backslash 0\ 10\ 12\ 6\ 10\ \backslash 1\ \backslash 1\ 6\\
6 &\rightarrow \backslash 1\ 8\\
8 &\rightarrow \backslash 0\ \backslash 1\\
10 &\rightarrow \backslash 0\ 8\\
12 &\rightarrow 6\ 8
\end{aligned}$$

In the format of IRRMGP the rules are numbered so the actual string characters are indicated by a single '\'. To avoid confusion in case these are also numbers in the inputted string. Then it is inputted into our algorithm and we get the following set of steps, building the transducer $T$:
Step 1: Rule $r_0 = 0 \rightarrow \backslash 0\ 12\ \backslash 0\ 10\ 12\ 6\ 10\ \backslash 1\ \backslash 1\ 6$ is processed. So $N_0 = 0$, $\eta_0 = \backslash 0\ 12\ \backslash 0\ 10\ 12\ 6\ 10\ \backslash 1\ \backslash 1\ 6$ and $T$ is as in Figure 3.1.

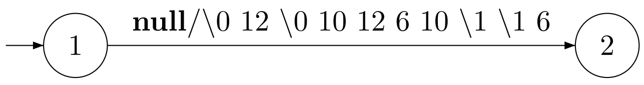

Figure 3.1: Transducer $T$ after processing rule $r_0$ of the grammar and minimisation.

Step 2: Rule $r_1 = 12 \rightarrow 6\ 8$ is processed. So $N_1 = 12$, $\eta_1 = 6\ 8$ and $T$ becomes as in Figure 3.2.

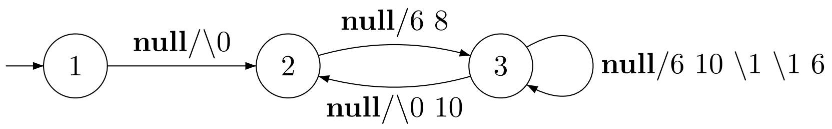

Figure 3.2: Transducer $T$ after processing rule $r_1$ of the grammar and minimisation.

Step 3: Rule $r_2 = 10 \rightarrow \backslash 0\ 8$ is processed. So $N_2 = 10$, $\eta_2 = \backslash 0\ 8$ and $T$ becomes as in Figure 3.3.

Step 4: Rule $r_3 = 6 \rightarrow \backslash 1\ 8$ is processed. $N_3 = 10$, $\eta_3 = \backslash 1\ 8$ and $T$ becomes as in Figure 3.4.

Step 4: Rule $r_4 = 8 \rightarrow \backslash 0\ \backslash 1$ is processed. $N_4 = 8$, $\eta_4 = \backslash 0\ \backslash 1$ and $T$ becomes as in Figure 3.5.

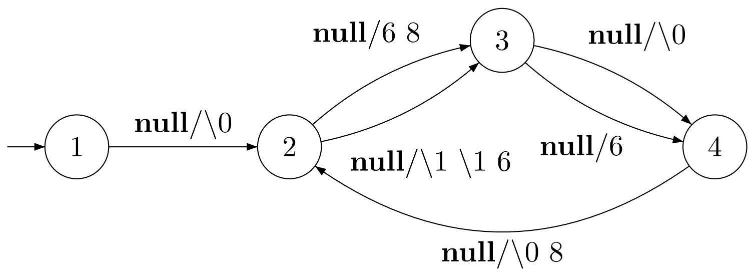

Figure 3.3: Transducer $T$ after processing rule $r_2$ of the grammar and minimisation.

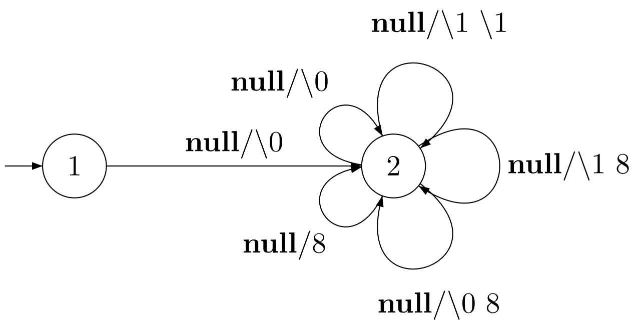

Figure 3.4: Transducer $T$ after processing rule $r_3$ of the grammar and minimisation.

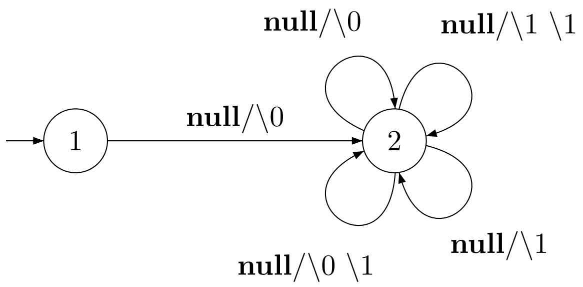

Figure 3.5: Transducer $T$ after processing rule $r_3$ of the grammar and minimisation.

Step 5: Figure 3.6 is the result of making $T$ deterministic and complete.

The algorithm's output is the $S_1$ encoded pair

$$\begin{aligned}(T,p) &= (1011 \cdot 0101 \cdot 1010 \cdot 0100 \cdot 10000 \cdot 0100 \cdot 10001 \cdot 0100 \cdot 1011 \\ &\quad \cdot 00100000 \cdot 1011 \cdot 010110110110 \cdot 01011 \cdot 01110, \\ &\quad 01011110101010011010011010100001 1).\end{aligned}$$

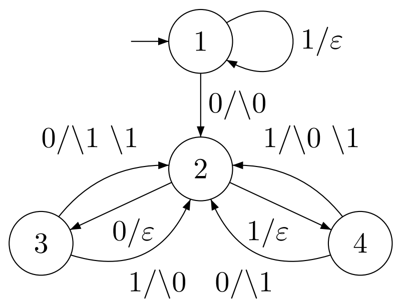

Figure 3.6: Transducer $T$ after processing all the rules of the grammar and then making it deterministic and complete.

In order to make the string more readable, we have decomposed the encoding of $T$ by concatenating each segment. The resulting pair is greater in size than the identity pair. Therefore we have not found a better upper bound for this sequence. Nevertheless, DNA sequences are much longer strings so the chances of finding a shorter description increase with the length of the string of interest $x$.

### 3.1.3 Bottom-Up Approach

As described in Subsection 3.1.1, this algorithm follows the concept of the top-down algorithm very closely but it reverses the order of the rule processing approach. In contrast to the previous approach in algorithm 3.1.1, in this algorithm we order the rules of the grammar according to the bottom-up ordering we previously defined. We suspect this allows for a more efficient algorithm as we can directly avoid 'overbuilding' the transducer, thus lessening the need to minimise. This is possible since we start the construction of the transducer from the most called rule, so the rule whose nodes will have greater in- and out-degrees in the non-deterministic version of the transducer. We therefore lose the need to build unnecessary nodes. The completed development of this algorithm is in the works.

**Algorithm 3.1.2:** BOTTOM-UP SLG TO T$(G, x)$

```
main
 $G \leftarrow$ REVERSEHIERARCHICALORDER($G$)
 $q_0 \leftarrow$ CREATENEWSTATE()
 $T \leftarrow (\{q_0\}, \Sigma, q_0, \Delta)$
 for each $r \in R$
   do RULEPROCESSING($r, S$)
 MAKEDETERMINISTIC()
 $control \leftarrow$ FINDCONTROLSEQ($x$)
 return $(T, control)$
```

**Brief:** This procedure manages the ordering of the rules of the grammar according to their appearance in the *rhs* of the rules. It pre-processes the grammar $G$ to initialise the required objects, calling the PARTIALORDER() procedure on those objects and rearranging the grammar according to the inverse of the partial order. Then it returns the bottom-up ordered grammar. Note that we assume $r_0$ (first encountered rule in $R$) to be the rule associated to $S$.

```
procedure HIERARCHICALORDER($G$)
 for each $r \in R$
    do { for each $N \in V$
       {   do if ISSUBSTRING(RHS($r$), $N$)
       {   then ADD($children, N$)
       { $node \leftarrow$ (LHS($r$), $children$)
       { ADD($nodeArray, node$)
       { $n = 0$
       { ADD($partialOrdering$, (LHS($r$), $n$))
 $partialOrdering \leftarrow$ PARTIALORDER($partialOrdering, nodeArray$, LHS($r_0$), 0)
 $G \leftarrow$ INVERSEMAP($G, partialOrder$)
 return ($G$)
```

**Brief:** Builds the partial ordering of the rules recursively.

```
procedure PARTIALORDER($partialOrdering, nodeArray, node, orderValue$)
 $n \leftarrow$ GETPARTIALORDERVALUE($partialOrdering, node$)
 $n \leftarrow$ MAX($n, orderValue$)
 $partialOrdering \leftarrow$ SETPARTIALORDERVALUE($partialOrdering, node, n$)
 $children \leftarrow$ GETCHILDREN($nodeArray, node$)
 $m \leftarrow n + 1$
 for each $c \in children$
   do $partialOrdering \leftarrow$ PARTIALORDER($partialOrdering, nodeArray, c, m$)
 return ($partialOrdering$)
```

**Brief:** Processes each rule $r$ by creating a new transition $t$ labelled $\eta$ (the *rhs* of $r$). Then, it searches through $t$ for any occurrences of every previously processed rules. If it finds any, these are linked into $t$ to form a new path, without repetitions. Finally, it induces state minimisation.

```
procedure ruleProcessing($r, S$)
 $N \leftarrow$ lhs($r$)
 $\eta \leftarrow$ rhs($r$)
 $q_t \leftarrow$ createNewState()
 if $N = S$
   then $q_s \leftarrow q_0$
   else $q_s \leftarrow$ createNewState()
 $t \leftarrow (q_s, \textbf{null}, q_t, \eta)$
 add($\Delta, t$)
 for each $P \in processed$
   do $\begin{cases} p_s \leftarrow \text{getSourceState}(repeatPaths, P) \\ p_t \leftarrow \text{getTargetState}(repeatPaths, P) \\ \text{findAndReplaceAllOccurrencesInT}(P, p_s, p_t, t) \end{cases}$
 add($repeatPaths, (N, q_s, q_t)$)
 add($processed, N$)
 minimiseStates()
```

**Brief:** Searches and replaces every occurrence of $P$ in the transition $t$, by linking in the existing corresponding path; whose source and target states are $p_s$ and $p_t$.

```
procedure findAndReplaceAllOccurrencesInT($P, p_s, p_t, t$)
 $e \leftarrow t$
 while isSubstring(getLabel($e$), $P$)
   then
     $label \leftarrow$ getLabel($e$)
     $pos \leftarrow$ findSubstring($label, P, 0$)
     $end \leftarrow pos + |P|$
     if $pos = 0$ and $end = |label|$
       then
         merge($p_s$, getSourceState($e$))
         merge($p_t$, getTargetState($e$))
         remove($\Delta, e$)
         break
     else if $pos = 0$ and $end \neq |label|$
       then
         merge($p_s$, getSourceState($e$))
         $target \leftarrow$ getTargetState($e$)
         $out \leftarrow$ getSubstring($label, end, |label|$)
         $f \leftarrow (p_t, \textbf{null}, target, out)$
         add($\Delta, f$)
         remove($\Delta, e$)
         $e \leftarrow f$
     else if $pos \neq 0$ and $end = |label|$
       then
         merge($p_t$, getTargetState($e$))
         $source \leftarrow$ getSourceState($e$)
         $out \leftarrow$ getSubstring($label, 0, pos$)
         $f \leftarrow (source, \textbf{null}, p_s, out)$
         add($\Delta, f$)
         remove($\Delta, e$)
         $e \leftarrow f$
     else if $pos \neq 0$ and $end \neq |label|$
       then
         $source \leftarrow$ getSourceState($e$)
         $target \leftarrow$ getTargetState($e$)
         $out1 \leftarrow$ getSubstring($label, 0, pos$)
         $out2 \leftarrow$ getSubstring($label, end, |label|$)
         $f \leftarrow (source, \textbf{null}, p_s, out1)$
         $g \leftarrow (p_t, \textbf{null}, target, out2)$
         add($\Delta, f$)
         add($\Delta, g$)
         remove($\Delta, e$)
         $e \leftarrow g$
```

**Brief:** Minimises the number of states by searching for unreachable and dead-end states and deleting them accordingly.

```
procedure minimiseStates()
 for each $q \in Q$
   do
     if inDegree($q$) $= 0$
       then
         for each $e \in \Delta$
           do
             if getSourceState($e$) $= q$
               then Remove($\Delta, e$)
         Remove($Q, q$)
 for each $q \in Q$
   do
     if outDegree($q$) $= 0$
       then
         for each $e \in \Delta$
           do
             if getTargetState($e$) $= q$
               then
                 $s \leftarrow$ getSourceState($e$)
                 $in \leftarrow$ getInputSymbol($e$)
                 $out \leftarrow$ getLabel($e$)
                 $f \leftarrow (s, in, s, out)$
                 add($\Delta, f$)
                 Remove($\Delta, e$)
         Remove($Q, q$)
```

**Brief:** Calls for an out-degree fix and completes each transition with a corresponding alphabet symbol, making the transducer complete and deterministic.

```
procedure makeDeterministic()
 fixOutDegree()
 for each $q \in Q$
   do
     $a \in \Sigma$
     for each $t$ such that getSourceState($t$) $= q$
       do
         remove($\Delta, t$)
         $target \leftarrow$ getTargetState($t$)
         $t \leftarrow (target, a, target,$ getLabel($t$)$)$
         add($\Delta, t$)
         $a \leftarrow$ nextSymbol($\Sigma$)
```

**Brief:** The following guarantees that the out-degree of every state is $|\Sigma|$, again traversed using a BFS, hidden for clarity's sake.

```
procedure fixOutDegree()
 for each $q \in Q$
   do
     while $(\textsc{outDegree}(q) - 2) > 0$
       do
         $p \leftarrow \textsc{createNewState}()$
         $count = 0$
         $transitionPair \leftarrow \textbf{null}$
         for each $t$ such that $\textsc{getSourceState}(t) = q$
           do
             if $count < 2$
               then
                 $\textsc{add}(transitionPair, t)$
                 $count \leftarrow count + 1$
               else break
         for each $t \in transitionPair$
           do
             $e \leftarrow (p, \textbf{null}, \textsc{getTargetState}(t), \textsc{getLabel}(t))$
             $\textsc{add}(\Delta, e)$
             $\textsc{remove}(\Delta, e)$
         $s \leftarrow (q, \textbf{null}, p, \varepsilon)$
         $\textsc{add}(\Delta, s)$
     while $\textsc{outDegree}(q) < 2$
       do
         $e \leftarrow (q, \textbf{null}, q, \varepsilon)$
         $\textsc{add}(\Delta, e)$
```

**Brief:** Traverses $T$ and—in a reverse engineering (or backtracking) way—obtains the corresponding *control* sequence to $x$, such that $T(control) = x$.

```
procedure findControlSeq($x$)
 $index \leftarrow 0$
 for each $q \in Q$
   do
     for each $t$ such that $\textsc{getSourceState}(t) = q$
       do
         $label \leftarrow \textsc{getLabel}(t)$
         if $(index + |label|) < |x|$
           then
             if $\textsc{isSubstring}(x, label, index)$
               then
                 $control \leftarrow control \cdot \textsc{getInputSymbol}(t)$
                 $index \leftarrow index + |label|$
 return $(control)$
```

## 3.2 Heuristics as an Optimisation Tool

Since we are strongly dependent on the resulting smallest grammar for our algorithms, and since our goal is to approximate the finite-state complexity of DNA sequences, it makes sense to target the construction of that grammar towards a reduction of the size of the resulting transducer-control pair. The smallest grammar problem is NP-complete as a decision problem [19]. Hence, solving the smallest grammar problem already requires a heuristics approach approximating the result and our idea of targeting, or guiding, the construction of the grammar to better our approximation seems natural. The state-of-the-art research on this problem focus their heuristics on pattern recognition, in order to better compress the string of interest and increase their chances of finding the smallest grammar for that string. In the case of DNA, such pattern recognition methods used are based certain types of patterns such as: maximal repeat, longest repeat and many others [15, 19]. In this section our task is to define a new heuristic, or score function, which improves the size of the resulting pair instead of the immediate resulting grammar. This section therefore describes the process to devise such a score function and its practical usage.

### 3.2.1 Devising a Score Funtion

In [15], the score functions are based on the number of occurrences of a chosen word (or “repeat” as they are called in Bioinformatics) and its length, in order to better reduce the size of the resulting straight-line grammar. Similarly, based on the number of occurrences of a word $w$ in the string of interest, $\alpha(w)$, and its length $|w|$, we formulate a score function in order to measure whether it is more advantageous to keep the word or to replace it by a non-terminal, and thus forming a new rule in the grammar.
Our transducer is essentially a digraph with an exact out-degree of 2. From graph theory, we know that a digraph has a number of arcs of

$$|A| = \sum_{v \in V} deg^{+}(v) = \sum_{v \in V} deg^{-}(v). \tag{3.1}$$

The above formula (3.1) states that the overall in-degree is equal to the overall out-degree. Thus we do not need to evaluate both. Here we will solely focus on the out-degree. It is a more complex process but it guarantees not to ‘forget’ any states added to the transducer as the only limitation is on the out-degree: $\forall v \in V, deg^{+}(v) = 2$. Hence, if the transducer has $n$ states, then the digraph has $n$ vertices and the number of transitions (or arcs) is $2n$. We do not know how many states there are. But we can deduce from our algorithms in Section 3.1 that for every word $w$, replacing $w$ with a non-terminal implies adding

$$2 \cdot \left\lceil \frac{\alpha(w)}{2} \right\rceil + 2 = 2 \cdot \left( \left\lceil \frac{\alpha(w)}{2} \right\rceil + 1 \right) \tag{3.2}$$

states.

Now, we need to measure the difference in the transducer encoding size before and after adding the states (and removing the word) in order to decide which is more beneficial to our goal. So we need to define what the encoding size of each word $w$ is with a 'direct' encoding and with a non-terminal replacement. We will assume that the encoding used is our $S_1$ encoding where each pair $(i, \text{string}(j))$ is encoded as $\text{bin}^{\#}(i+1) \cdot \text{string}^{\S}(j+1)$. The pair $(i, \text{string}(j))$ represents the target state $q_i$ and the output function (or the transition label, as referred to in the algorithms). If a word $w$ is replaced by a non-terminal in the grammar, we can measure the size of the new encoding. But it seems very difficult to measure what is 'taken off' or 'saved' from the current encoding. The only means we have to do this seems to be a direct comparison of the results, and this is costly. However, it is obvious that for all $i, j$ such that $i < j$, $|\text{string}^{\S}(i)| \leq |\text{string}^{\S}(j)|$. We then considered finding an approximation or an average measure of the difference between $|\text{string}^{\S}(i)|$ and $|\text{string}^{\S}(j)|$ based on the difference between $i$ and $j$, to finally use that as the score function (or weight) for each word in the sequence. Unfortunately, evaluating this is in fact a difficult and expensive task. The best we can do at this time is to give an upper bound. Otherwise we would have to simulate the conversion for each word for an exact measure of the difference.

Let *curr* be the current encoding. Let $S_i$, for $i = 1, \ldots, m$, be the sequences in which the word $w$ occurs, where $m$ is the number of such sequences, and let $u_j$, for $j = 1, \ldots, \alpha(w)+1$, be the words such that for each $1 \leq i \leq m$, $S_i = u_1 \cdot w \cdot u_2 \cdot w \cdots u_l$, where $l \leq \alpha(w)+1$ depending on whether there are more $u_j$s to be found in the other $S_i$s. Let us also define the function $\gamma : \Sigma^* \to \mathbb{N}$ such that for all $s \in \Sigma^*$, $\gamma(s)$ is the lexicographic index of $s$ in the enumeration of all strings in $\Sigma^*$. Hence the score function, $\text{score}_T : \Sigma^* \to \mathbb{N}$, measures the size of the encoding if we removed the $m$ $S_i$-sequences from *curr* and replaced those with one encoded occurrence of $w$, all the $u_l$ fragments of the $S_i$-sequences (not forgetting to add the new states and the $\varepsilon$-loops that complete the new version of the transducer). Formally, we obtain the following:

$$\begin{aligned}
\text{score}_T(w) &= |curr| - \sum_{i=1}^{m} \left|\text{string}^{\S}(\gamma(S_i)+1)\right| \\
&+ \left|\text{string}^{\S}(\gamma(w)+1)\right| + \sum_{j=1}^{\alpha(w)+1} \left|\text{string}^{\S}(\gamma(u_i)+1)\right| \\
&+ \left(2 \cdot \left(2 \cdot \left(\left\lceil \frac{\alpha(w)}{2} \right\rceil + 1\right) - m\right) - (\alpha(w)+2)\right) \cdot \left|\text{string}^{\S}(2)\right| \\
&+ \alpha(w) \cdot \left|\text{bin}^{\#}(|Q|+1)\right| + 2 \cdot \sum_{k=|Q|+2}^{\alpha(w)+|Q|} \left|\text{bin}^{\#}(k)\right|. \qquad (3.3)
\end{aligned}$$

Let us justify this score function. First of all, we need to substract the size of the encoded sequences which hold our target word $w$ as a substring since we want the new transducer to only have a single occurrence of $w$, in its own 'state-transduction-state' triple. Then, we need to add all the components which come with replacing this string with a non-terminal in the grammar. Hence, we add the size of the encoded strings which compose each of the previous sequences, but with only one occurrence of the encoding of $w$. We know that there are $\alpha(w)+1$ non-$w$ components to add. We also have to add all of the $\varepsilon$-labelled transitions. These labels are 'fill-ins' in order to make sure the transducer is deterministic and complete. So, given that the out-degree of each state is 2, there are twice the number of added states such transition labels. However, we mustn't forget that many target states in the new path are not in fact added. Those states already exist in the current transducer; they are each of the target states of the $m$ $S_i$-sequences. Moreover, of all the added transitions we already have $\alpha(w)+2$ accounted for. Thus, there are $\left(2\cdot\left(2\cdot\left(\left\lceil\frac{\alpha(w)}{2}\right\rceil+1\right)-m\right)-(\alpha(w)+2)\right)$ $\varepsilon$-labelled transitions. We have therefore covered all of the changes to the output function of the tranducer. Now we also need to cover the first projection of $\Delta$, the transition function of the transducer. We know that $w$ will be outputted $\alpha(w)$ times, so we need to add its corresponding state that many times. Finally, every other added state will have an expected in-degree of 2, hence why we need to add each of their 'pointer' twice.

A few issues arise from this and all related to one topic: isomorphism. Recall from Chapter 2 that the finite-state complexity is the size of the description of a given finite sequence. We cannot guarantee our resulting transducer will be the smallest, but how do we guarantee that—since the encoding is so index dependent—it will be the first encountered? Is it possible that even though we do have a correct approximation of the smallest transducer, the way that we encode it results in a non-optimal isomorphic version? And how do we correct the encoding in that eventuality?

**Open Problem 59.** *Is there a way to guarantee that the resulting approximate description of a string is co-lexicographically first?*

One can then wonder why the issue in the Open Problem 59 matters. The fact is that the algorithms we are dealing with here provide an approximation and hopefully a better upper bound to the complexity on really long strings. So, if we were to use these resulting pairs as a means to prune the search space for the actual shortest description, since the search is in lexicographical order, the 'position' of this resulting pair dramatically affects how much of the search space is pruned. In turn this affects the time complexity of this new search.

Nevertheless, with our current knowledge we can provide some properties about the encoding. From Chapter 1.3we know that

$$|\text{string}^{\S}(n)| = 2 \cdot \lceil \log_2 (\lceil \log_2(n) \rceil + 1) \rceil + \lceil \log_2(n) \rceil + 1. \tag{3.4}$$

So, we have, for $s \in \Sigma^*$ and for $u, v, w$ substrings of $s$ such that $s = u \cdot v \cdot w$,

$$\begin{aligned} \left|\text{string}^{\S}(\gamma(s))\right| &= 2 \cdot \lceil \log_2 (\lceil \log_2(\gamma(s)) \rceil + 1) \rceil + \lceil \log_2(\gamma(s)) \rceil + 1 \\ &\leq 2 \cdot \lceil \log_2 (\lceil \log_2(\gamma(u)) \rceil + 1) \rceil + \lceil \log_2(\gamma(u)) \rceil \\ &\quad +2 \cdot \lceil \log_2 (\lceil \log_2(\gamma(v)) \rceil + 1) \rceil + \lceil \log_2(\gamma(v)) \rceil \\ &\quad +2 \cdot \lceil \log_2 (\lceil \log_2(\gamma(w)) \rceil + 1) \rceil + \lceil \log_2(\gamma(w)) \rceil + 3 \\ &= \left|\text{string}^{\S}(\gamma(u))\right| + \left|\text{string}^{\S}(\gamma(v))\right| + \left|\text{string}^{\S}(\gamma(w))\right|. \end{aligned} \tag{3.5}$$

Also recall that $|\text{bin}^{\#}(n)| = |\text{string}^{\S}(n)|$ (see Table 1.3). Thus, if each of the components of $s$ occur only once, then it is disadvantageous to replace any of them with a non-terminal. But once we have some repetitions of substrings, we find that it is a favourable move. However, we suspect a threshold exists in the number of occurrences for which the advantage holds, as the amount of states—and with it the costly necessity to complete the transducer—also increases with that number. In order to have a better idea of what that threshold might be, we need to have at least an approximate value for the indexing function, $\gamma$. Thankfully, we know that the indexing is in co-lexicographic order (see Section 1.2), so we can calculate $\gamma$ exactly as follows:

$$\forall s \in \Sigma^*, \gamma(s) = \sum_{n=0}^{|s|-1} |\Sigma|^n + c_s + 1, \tag{3.6}$$

where $c_s$ is the lexicographic index of $s$ within all strings of size $|s|$, which is

$$c_s = (s)_{|\Sigma|}.$$

We now have a score function which will aim at minimising the resulting transducer built from the grammar. However, it does not consider the size of the control sequence, which is an integral part of the finite-state complexity measure. We need to modify the above score function and incorporate a control sequence measure. The length control sequence is the number of transitions needed to traverse to obtain the sequence of interest $x$. Therefore, we know that the control sequence augments with the number of occurrences of a word $w$ as that new transition will have to be used $\alpha(w)$ times. Moreover, every component will be used, so we also need to add $\alpha(w) + 1$ to the length of the control sequence. As a result, the increase of the length of the control sequence $p$ is measured by the following function:

$$\ell(p) = |p| + 2 \cdot \alpha(w) + 1. \tag{3.7}$$

Thus, when we combine the functions (3.3) with (3.7), we obtain:

$$\begin{aligned}
\mathrm{score}_{(T,p)}(w) &= |curr| - \sum_{i=1}^{m} \left|\mathrm{string}^{\S}\left(\gamma(S_i)+1\right)\right| \\
&+ \left|\mathrm{string}^{\S}\left(\gamma(w)+1\right)\right| + \sum_{j=1}^{\alpha(w)+1} \left|\mathrm{string}^{\S}\left(\gamma(u_i)+1\right)\right| \\
&+ \left(2\cdot\left(2\cdot\left(\left\lceil\frac{\alpha(w)}{2}\right\rceil+1\right)-m\right)-(\alpha(w)+2)\right)\cdot\left|\mathrm{string}^{\S}(2)\right| \\
&+ \alpha(w)\cdot\left|\mathrm{bin}^{\#}(|Q|+1)\right| + 2\cdot\sum_{k=|Q|+2}^{\alpha(w)+|Q|}\left|\mathrm{bin}^{\#}(k)\right| \\
&+ 2\cdot\alpha(w)+1, \qquad (3.8)
\end{aligned}$$

where *curr* now includes both the size of the current transducer and the size of the current control sequence. Initially, $|curr| = |\sigma_0|+1$, where $\sigma_0$ is the encoding of the machine in Figure 3.7 and where the control sequence is obviously a single bit (here '1').

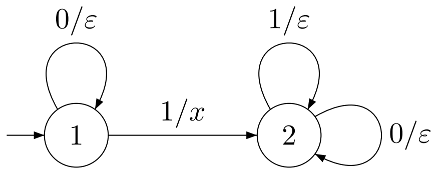

Figure 3.7: Transducer $T_{\sigma_0}$ representing the deterministic and completed first instance of the transducer in Algorithm 3.1.2, no matter the inputted grammar. It also represents the resulting transducer if for some string $x$, the grammar is a single start rule (i.e. no repetitive words are compressed).

According to the encoding $S_1$ that we are using, this transducer $T_{\sigma_0}$ is encoded as

$$\begin{aligned}
\sigma_0 &= \mathrm{bin}^{\#}(2)\cdot\mathrm{string}^{\S}(2)\cdot\mathrm{bin}^{\#}(3)\cdot\mathrm{string}^{\S}(\gamma(x)+1)\cdot\mathrm{bin}^{\#}(3)\cdot\mathrm{string}^{\S}(2) \\
&\quad\cdot\mathrm{bin}^{\#}(3)\cdot\mathrm{string}^{\S}(2) \\
&= 1010\cdot 0100\cdot 1011\cdot\mathrm{string}^{\S}(\gamma(x)+1)\cdot 1011\cdot 0100\cdot 1011\cdot 0100 \\
&= 101001001011\cdot\mathrm{string}^{\S}(\gamma(x)+1)\cdot 1011010010110100.
\end{aligned}$$

From (3.4) and (3.6), we know that

$$\begin{aligned}
|\sigma_0| &= |101001001011| + |\text{string}^{\S}(\gamma(x)+1)| + |1011010010110100| \\
&= |\text{string}^{\S}(\gamma(x)+1)| + 28 \\
&= 2 \cdot \lceil \log_2(\lceil \log_2(\gamma(x)+1) \rceil + 1) \rceil + \lceil \log_2(\gamma(x)+1) \rceil + 29 \\
&= 2 \cdot \left\lceil \log_2 \left( \left\lceil \log_2 \left( \sum_{n=0}^{|x|-1} |2|^n + (x)_2 + 2 \right) \right\rceil + 1 \right) \right\rceil \\
&+ \left\lceil \log_2 \left( \sum_{n=0}^{|x|-1} |2|^n + (x)_2 + 2 \right) \right\rceil + 29.
\end{aligned}$$

We therefore have all the necessary mathematical components to implement this score function, and begin to construct new straight-line grammars.

## 3.2.2 Using the Score Function

The score function was devised to be used as a preprocessing tool for our algorithms (hence in the construction of the grammar from the sequence of interest). Therefore, in collaboration with Matthias Gallé, we implemented this score function into his IRRMGP algorithm [26] such that we could produce some experimental results. These are still being collected. This way we have a means to empirically compare both the grammars as well as the transducer-control pair results with and without the use of this score function, which is meant to better our outcome.

**Conjecture 60.** *There exists an integer $M$ such that for all DNA sequences $s$ of length greater or equal to $M$, the resulting approximative finite-state complexity from Algorithm 3.1.2 using $\text{score}_T$ will be smaller than that using the standard maximal compression score function in IRRMGP.*

Another possibility which we have not explored in this thesis, is to use the score function in the algorithms themselves as a means to decide whether to use or convert a chosen non-terminal of the grammar. We suspect that only the top-down algorithm would truly be adaptable to such procedures; however the matters of efficiency and comparison of results between the top-down and the bottom-up approaches become relevant and interesting. This work could then evolve into a direct construction of the pair $(T, p)$ from the sequence of interest $x$, making it an independent computation from the grammars. However, there is some debate regarding the benefits and efficiency of this direct computation, as grammars are quite different from transducers even though context-free grammars are more powerful than finite automata. We leave this matter for future work.

## 3.3 Summary

Our general approach was to merge the two concepts described in the chapter's introduction by using the obtained smallest grammar and converting it into an equivalent minimal transducer with its corresponding input (or control sequence) to generate the string of interest. Since our goal is to approximate the finite-state complexity of DNA, we have chosen a grammar-compressor which has been built for that same domain, IRRMGP [15, 26]. This approach allows for a better approximation of the finite-state complexity of that string, in the case where the resulting pair is smaller than the upper bound given in Corollary 21. In particular, two approaches have been developed and are presented in this chapter: a top-down and a bottom-up approach. The 'direction' of the approach depends on the ordering of the process to convert the grammar; whether we work from the start rule (associated with the start variable) to the most referenced rule, or the other way around. Once these two algorithms were developed, we considered the idea of guiding the construction of the grammar in order to better the results of our algorithms. Subsequently, this brought forth the development of the score function in Section 3.2. It seemed natural to create a new heuristic for building the SLG which would aim at reducing the size of the resulting transducer, since the grammar is an intermediate step of our process.

The first item on our future project's list is to obtain a thorough set of experimental results on transposons, a set which is currently being populated but too insignificantly small to be in anyway conclusive at this stage. These empirical results will allow us to conclusively prove or disprove Conjecture 60. As we have discussed in this chapter, future work would include developing procedures to include the score function in the conversion algorithms, to see if those would more efficiently reduce the size of the pair or not. We could also adapt the work and test for a further reduction of the pair with the consideration of common prefixes in out-degree transition (or transduction) outputs; an idea inspired by the Onward Subsequential Transducer Inference Algorithm (OSTIA) as presented in [31].

OSTIA is an algorithm which infers a subsequential transducer from a training set of input-output pairs for the desired transducer. It does so by first constructing a tree transducer from the training set. Then the output symbols are transferred closer to the root of the tree, which becomes the initial state of the transducer. The transfer is based on recognising the longest common prefixes of the outputs in the branches and 'pushing' those prefixes back towards the root.

When considering the prefixes, we could choose in our case, in a local or global manner, which pairs of transitions to merge or to link to new intermediate states. This will get rid of unnecessary $\varepsilon$-labelled transitions and possibly minimise the size of the encoding.

This also leads to another study: that of minimising the ambiguity. This concept could in turn be used in grammars, especially grammars with advice. We have also wondered whether this idea could be used to alter the above score function or create new heuristics on a local (as a greedy approach) or a global (towards an optimal score) scale of ambiguity measure. But, for further research and studies, we are particularly interested in bypassing the grammar and in exploring the possibility of a direct construction of the transducer-control pair from the sequence itself. The forner would definitely provide a fuller, more compelling exploitation of the computability of finite-state complexity.

Chapter 4

# Conclusions and Future Work

In this thesis we have proposed a computable counterpart to Kolmogorov complexity based on finite-state transducers: the finite-state complexity. We also attempted to apply this new complexity measure to the domain of DNA compression. Several other approaches were previously offered to avoid the incomputability of all the descriptional complexities, also working with simpler machines than Turing machines such as context-free grammars or finite automata. Finite transducers have the combined advantage of being a specialised type of finite automaton and dealing with an input-output pair of strings, more naturally imitating Turing machines. They are also enumerable as a regular set of encodings, as we proved in Chapter 1. This new complexity is the fundamental basis towards a new variant of AIT and we provided many promising results.

In Chapter 2 we presented, in detail, the finite-state complexity and its theoretical, computational and practical results. In particular, we proved that the finite-state complexity is computable and satisfies the Invariance Theorem. Furthermore, we exploited its computability and extracted some results relating to its capability (or lack thereof) to satisfy a strong form of the Borel normality and FS-incompressibility correlation. We also analyzed the subject of state-size hierarchy and our main result in the matter was that it is infinite with respect to the standard as well as any 'natural' encoding.

In Chapter 3 we provided the beginnings of a practical application of the finite-state complexity with the aim of applying it to DNA compression. We devised a pair of algorithms approximating the complexity of a DNA sequence, given some smallest SLG for that same sequence. We furthered the work by formulating a dedicated score function, which builds the smallest SLG according to the size of the resulting transducer.

Essentially, the future work is listed in the many conjectures and open problems that have yet to be decided upon. Our insights and questions on the exact place the finite-state complexity holds in the NP hierarchy were expressed in Conjecture 23 and Open Problem 24.

Our interest in Borel normality and its relation to FS-incompressibility was made clear through Conjectures 40 and 43, as well as in Open Problems 33 and 44. These questions are definitely prioritised in our future works as a positive theoretical answer would significantly strengthen the new theory.

Moreover, Open Problem 45 represents our growing enthusiasm in discovering the behaviour of the finite-state complexity as we tend towards infinity and work in that domain is definitely needed.

In the area of state-size hierarchy, Conjectures 50 and 57 remain to be proven. In addition, many compelling open problems on the state-size hierarchy are yet to be answered: Open Problems 51, 52, 53 and as well as various word-related combinatorial problems.

Finally, we also have Conjecture 60 and Open Problem 59 to resolve concerning our practical work. Notice however that the algorithms devised, as well as the score function, do not explicitly depend on the DNA sequence. Hence we could easily apply this work to a wider range of strings or we could make better use of the DNA features to optimise them.

# Appendices

# Appendix A

# Finite-State Complexity Plots

This appendix holds a set of plots to depict the convergence, even on these early results, of the complexities based on our $S_0$ and $S_1$ encodings. They also depict how much more compact $C_{S_1}$ is and we can start to see how, as we tend towards infinity, $C_{S_1}$ will obtain smaller complexity measures than $C_{S_0}$.

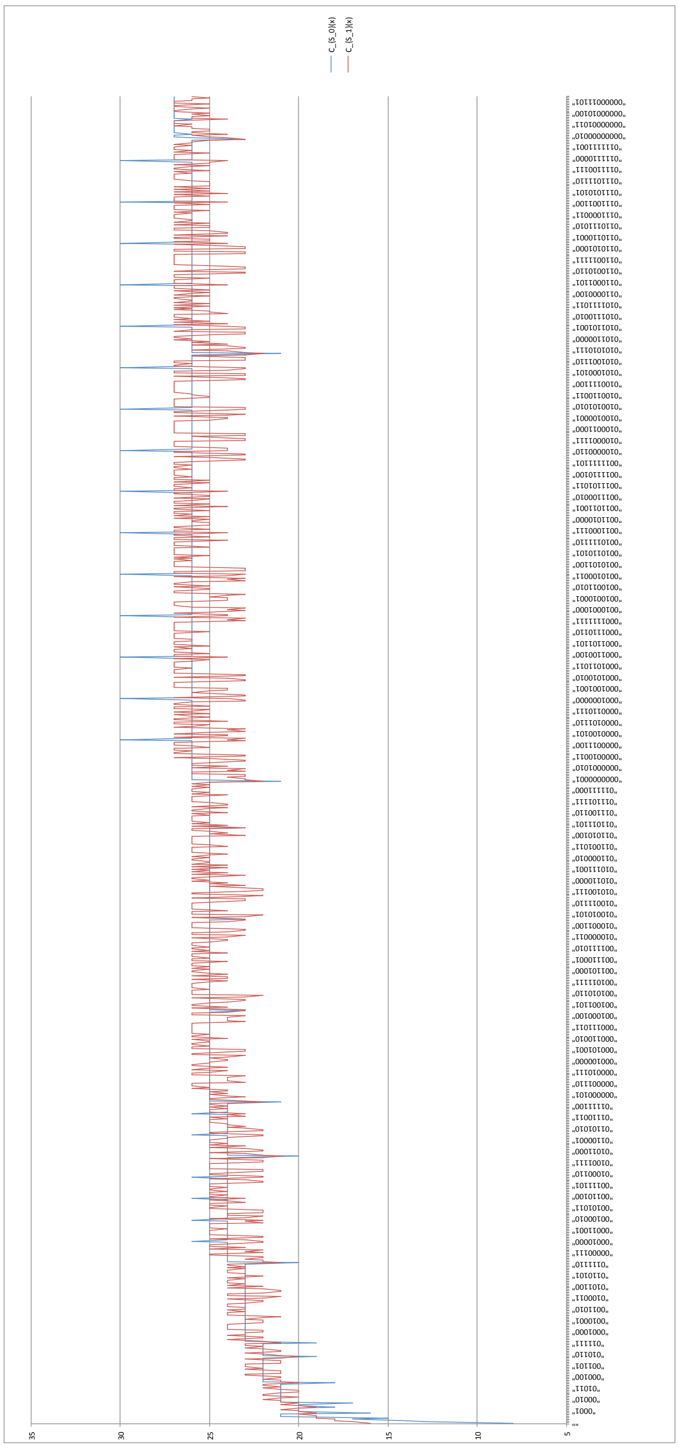

Figure A.1: Plot of comparative data in Tables 2.1 and 2.2. Note that there are over 1000 strings in the data set, and they could not all fit on the x-axis. The data is all the strings from $\varepsilon$ to 0100100011. The complexity measures between $C_{S_0}$ (in blue) and $C_{S_1}$ (in red) vary from 0 to 8 and range over $[8, 30]$.

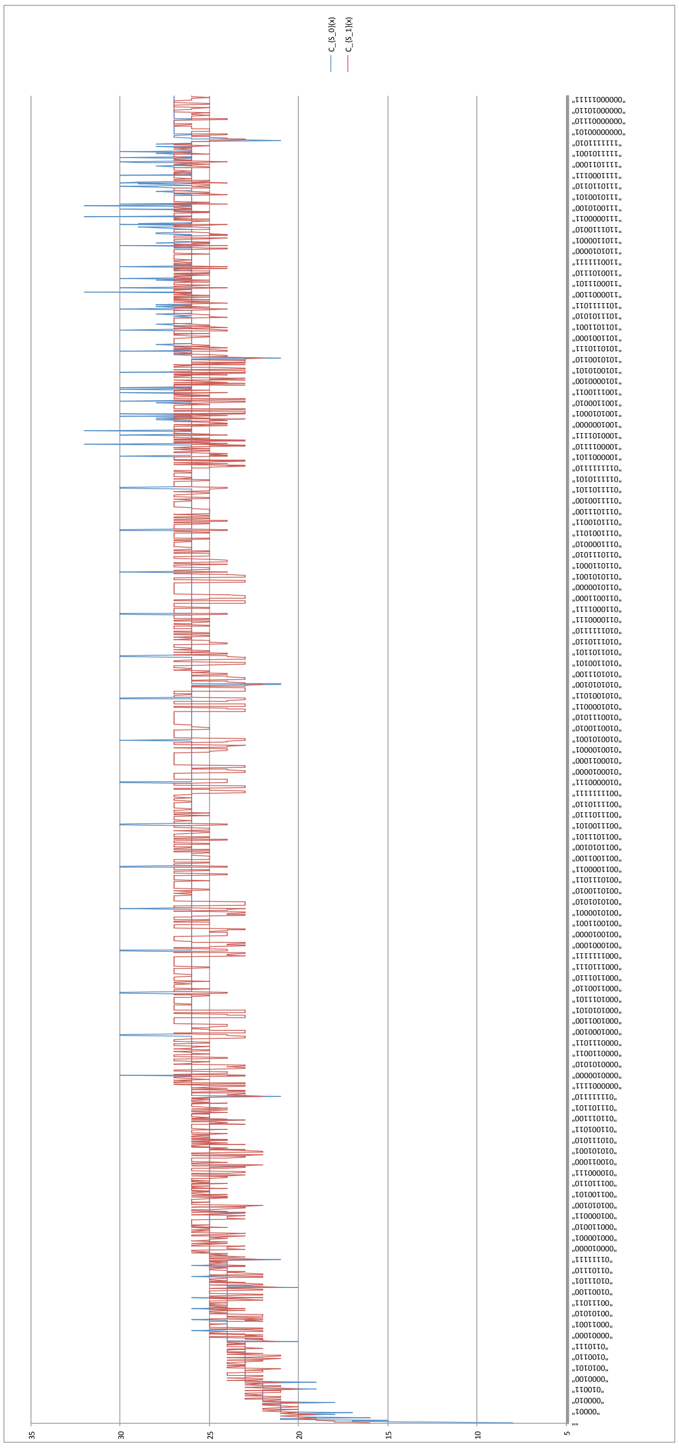

Figure A.2: More complete plot of the comparative data in Tables 2.1 and 2.2. Note that there are over 2000 strings in the data set, and they could not all fit on the x-axis. The data is all the strings from $\varepsilon$ to 00000100010. The complexity measures between $C_{S_0}$ and $C_{S_1}$ vary from 0 to 8. The range of complexity measures is $[8, 32]$. $C_{S_0}$ is depicted in blue, $C_{S_1}$ in red.

# Bibliography

[1] V. N. Agafonov. Normal sequences and finite automata. *Soviet Mathematics Doklady*, 9:324–325, 1968. [cited at p. 2, 5]

[2] J.-P. Allouche and J. Shallit. *Automatic Sequences.* Cambridge University Press, 2003. [cited at p. 2]

[3] I. Althöfer. Tight lower bounds on the length of word chains. *Information Processing Letters*, 34:275–276, 1990. [cited at p. 23]

[4] S. Arora and B. Barak. *Computational Complexity: A Modern Approach.* Cambridge University Press, 2009. [cited at p. 4]

[5] J. Arpe and R. Reischuk. On the complexity of optimal grammar based compression. In *Proceedings of the Data Compression Conference, DCC'06*, pages 173–186, 2006. [cited at p. 16]

[6] J. Berstel. *Transductions and Context-free Languages.* Teubner, 1979. [cited at p. 3, 6, 14, 20]

[7] C. Bourke, J. M. Hitchcock, and N. V. Vinodchandran. Entropy rates and finite-state dimension. *Theoretical Computer Science*, 3(349):392–406, 2005. [cited at p. 2, 5]

[8] H. Buhrman and L. Fortnow. Resource-bounded Kolmogorov complexity revisited. In *Proceedings STACS'97*, Lectures Notes in Computer Science 1200, pages 105–116. Springer, 1997. [cited at p. 1]

[9] C. S. Calude. Borel normality and algorithmic randomness. *Developments in Language Theory*, pages 113–129, 1994. [cited at p. 5, 29]

[10] C. S. Calude. *Information and Randomness: An Algorithmic Perspective.* Springer, Berlin, 2nd edition, 2002. [cited at p. 1, 3, 7, 14]

[11] C. S. Calude, N. J. Hay, and F. C. Stephan. Representation of left-computable $\varepsilon$–random reals. *Journal of Computer and System Sciences*, 2010. DOI: 10.1016/j.jcss.2010.08.0001. [cited at p. 14]

[12] C. S. Calude, K. Salomaa, and T. K. Roblot. Finite-state complexity and randomness. *CDMTCS Research Report*, (374), 2009. Revised June 2010. [cited at p. iii, 2, 13, 34, 35, 40, 42]

[13] C. S. Calude, K. Salomaa, and T. K. Roblot. Finite-state complexity and the size of transducers. *EPTCS*, 31:38–47, August 2010. [cited at p. iii, 2, 42, 43]

[14] C. S. Calude, K. Salomaa, and T. K. Roblot. State-size hierarchy for FS-complexity. *International Journal of Foundations of Computer Science*, submitted November 2010. [cited at p. iii, 2, 43]

[15] R. Carrascosa, F. Coste, M. Gallé, and G. Infante-Lopez. The smallest grammar problem as constituents choice and minimal grammar parsing. *Journal of Computer and Systems Sciences*, September 2010. [cited at p. 3, 22, 45, 46, 63, 69]

[16] G. J. Chaitin. On the simplicity and speed of programs for computing infinite sets of natural numbers. *Jounal of the ACM*, 16(407), 1969. [cited at p. 4]

[17] G. J. Chaitin. *Algorithmic Information Theory.* Cambridge University Press, 1987. [cited at p. 1, 4, 5]

[18] G. J. Chaitin. *Information, Randomness and Incompleteness: papers on algrithmic information theory.* World Scientific, Singapore, 2nd edition, 1990. [cited at p. 4]

[19] M. Charikar, E. Lehman, A. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2002, revised 2005. [cited at p. 45, 63]

[20] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Rasala, A. Sahai, and A. Shelat. Approximating the smallest grammar: Kolmogorov complexity in natural models. In *Proceedings of STOC'02*, pages 792–801. ACM Press, 2002. [cited at p. 1, 2, 22]

[21] F. Coste and T. K. Roblot. Towards an evaluation of the finite-state complexity of dna. Technical report, INRIA, 2010. [cited at p. iii, 2, 46]

[22] N. de Bruijn. A combinatorial problem. In *Proceedings of the Koninklijke Nederlandse Akademie Van Wetenschappen (KNAW)*, number 49, pages 758–764, 1946. [cited at p. 23]

[23] M. Domaratzki, G. Pighizzini, and J. Shallit. Simulating finite automata with context-free grammars. *Information Processing Letters*, 84:339–344, 2002. [cited at p. 22, 23]

[24] D. Doty and P. Moser. Feasible depth. In *Computation and Logic in the Real World*, volume 4497, pages 228–237. Computability in Europe (CiE), Spring-Verlag, 2007. [cited at p. 2]

[25] R. Downey and Hirschfeldt D. *Algorithmic Randomness and Complexity.* Springer, Heilderberg, 2010. [cited at p. 1]

[26] M. Gallé. *Searching for Compact Hierarchical Structures in DNA by means of the Smallest Grammar Problem.* PhD thesis, Université de Rennes 1, 2011. [cited at p. 68, 69]

[27] R.K Guy. *Unsolved Problems in Number Theory.* Springer, Berlin, 3rd edition, 2004. [cited at p. 36]

[28] E. Lehman. *Approximation algorithms for grammar-based compression.* PhD thesis, MIT, 2002. [cited at p. 1, 16]

[29] E. Lehman and A. Shelat. Approximation algorithms for grammar-based compression. In *SODA'02*, pages 205–212. SIAM Press, 2002. [cited at p. 1, 22]

[30] A. Nies. *Computability and Randomness.* Clarendon Press, Oxford, 2009. [cited at p. 1, 4]

[31] J. Oncina, P. Gracía, and E. Vidal. Learning subsequential transducers for pattern recognition interpretation tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 3(13):252–264, March 1991. [cited at p. 69]

[32] E. Rivals, J.-P. Delahaye, M. Dauchet, and O. Delgrange. A guaranteed compression scheme for repetitive dna sequences. Technical report, Université des Sciences et Technologies de Lille, November 1995. [cited at p. 45]

[33] T. K. Roblot. Finite-state descriptional complexity. Honours' Dissertation, University of Auckland, 2009. [cited at p. iii, 2, 13, 26]

[34] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302:211–222, 2002. [cited at p. 1, 22]

[35] C. P. Schnorr and H. Stimm. Endliche automaten und zufallsfolgen. *Acta Informatica*, 1:345–359, 1972. [cited at p. 2, 5]

[36] J. Shallit. The computational complexity of the local postage stamp problem. *SIGACT News*, 33:90–94, 2002. [cited at p. 36, 37]

[37] J. Shallit. What this country needs is an 18 cent piece. *Mathematical Intelligencer*, 25:20–23, 2003. [cited at p. 36]

[38] J. Shallit and Y. Breitbart. Automacity i: Properties of a measure of descriptional complexity. *Journal of Computer and System Sciences*, 53:10–25, 1996. [cited at p. 2]

[39] J. Shallit and M.-W. Wang. Automatic complexity of strings. *Journal of Automata, Languages, and Combinatorics*, 6:537–554, 2001. [cited at p. 2, 19]

[40] M. Sipser. *Introduction to the Theory of Computation.* Course Technology, second edition, 2005. [cited at p. 2, 3, 4, 5, 7, 14]

[41] M. Trenton. Some results on Borel-normal strings. Technical Report 2, University of Prince Edward Island, November 2006. [cited at p. 29]

[42] A. Turing. On computable numbers, with an application to the entscheidungs problem. In *Proceedings of the London Mathematical Society*, volume 42 of *2*, pages 230–265, TMs, AIT, halting pb 1936. [cited at p. 5]

[43] J.H. van Lint and R.M. Wilson. *A course in combinatorics.* Cambridge University Press, 1993. [cited at p. 23]

[44] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24:530–536, 1978. [cited at p. 2, 5]