UNIVERSITÀ DEGLI STUDI DI MILANO

DIPARTIMENTO DI INFORMATICA

PH.D. IN COMPUTER SCIENCE — XXXII CYCLE

Luca Prigioniero

Regular Languages:

To Finite Automata and Beyond

Succinct Descriptions and Optimal Simulations

DOCTORAL THESIS

SUPERVISED BY

Prof. Giovanni Pighizzini

ACADEMIC YEAR 2018/2019

Contents

Abstract

It is well known that regular — or type 3 — languages are equivalent to finite automata. Nevertheless, many other characterizations of this class of languages in terms of computational devices and generative models are present in the literature. For example, by suitably restricting more general models such as context-free grammars, pushdown automata, and Turing machines, that characterize wider classes of languages, it is possible to obtain formal models that generate or recognize regular languages only. The resulting formalisms provide alternative representations of type 3 languages that may be significantly more concise than other models that share the same expressing power.

The goal of this work is to investigate these formal systems from a descriptional complexity perspective, or, in other words, to study the relationships between their sizes, namely the number of symbols used to write down their descriptions. We also present some results related to the investigation of the famous question posed by Sakoda and Sipser in 1978, concerning the size blowups from nondeterministic finite automata to two-way deterministic finite automata.

Abstract

È noto che i linguaggi regolari — o di tipo 3 — sono equivalenti agli automi a stati finiti. Tuttavia, in letteratura sono presenti altre caratterizzazioni di questa classe di linguaggi, in termini di modelli riconoscitori e grammatiche. Per esempio, limitando le risorse computazionali di modelli più generali, quali grammatiche context-free, automi a pila e macchine di Turing, che caratterizzano classi di linguaggi più ampie, è possibile ottenere modelli che generano o riconoscono solamente i linguaggi regolari. I dispositivi risultanti forniscono delle rappresentazioni alternative dei linguaggi di tipo 3, che, in alcuni casi, risultano significativamente più compatte rispetto a quelle dei modelli che caratterizzano la stessa classe di linguaggi.

Il presente lavoro ha l'obiettivo di studiare questi modelli formali dal punto di vista della complessità descrizionale, o, in altre parole, di analizzare le relazioni tra le loro dimensioni, ossia il numero di simboli utilizzati per specificare la loro descrizione. Sono presentati, inoltre, alcuni risultati connessi allo studio della famosa domanda tuttora aperta posta da Sakoda e Sipser nel 1978, inerente al costo, in termini di numero di stati, per l'eliminazione del nondeterminismo dagli automi stati finiti sfruttando la capacità degli automi two-way deterministici di muovere la testina avanti e indietro sul nastro di input.

Introduction

The investigation of computational models operating under restrictions is one of classical topics of computer science.

In one of his pioneer papers, Chomsky introduced a hierarchy of classes of languages, also known as *Chomsky hierarchy*, obtained by applying increasing restrictions to general *grammars*, that characterize the class of type 0 languages [Cho59a]. In this way he introduced the classes of *context-sensitive* (or type 1), *context-free* (or type 2), and *regular* (or type 3) languages.

For the same classes of languages, there also exist characterizations in terms of *computational devices*. Even in this case, bounding computational resources of general models, less powerful devices can be obtained. For example, while nondeterministic Turing machines (even in the one-tape version) characterize type 0 languages, by restricting the working space they are allowed to use to the portion of the tape that initially contains the input, linear bounded automata are obtained, that are equivalent to context-sensitive languages [Kur64]. Also finite automata or pushdown automata, that are standard recognizers for type 3 and type 2 languages, respectively, can be considered as particular Turing machines in which the access to the memory storage is limited.

Besides the standard models mentioned so far, considering machines that make restricted use of resources, it is possible to obtain alternative characterization of the classes

of the hierarchy. For example, in 1965, Hennie proved that when the length of the computations, *i.e.*, the time, is linear in the input length, one-tape Turing machines are no more powerful than finite automata, that is, they recognize regular languages only [Hen65].

As remarked by Chomsky, context-free languages have the property of being able to describe recursive structures such as, for instance, nested parentheses, arithmetic expressions, and typical programming language constructs. In terms of recognizing devices, this capability is implemented through the pushdown store, which is used to add recursion to finite automata, so making the resulting model, namely pushdown automata, equivalent to context-free grammars [Cho62].

To emphasize the ability of context-free grammars to generate recursive sentential forms, Chomsky investigated the *self-embedding* property [Cho59b]: a context-free grammar is self-embedding if it contains a variable which, in some sentential form, is able to reproduce itself surrounded by two nonempty strings. Roughly speaking, this means that such a *self-embedded* variable is "truly" recursive. He proved that, among all context-free grammars, only self-embedding ones can generate nonregular languages. Hence, *non-self-embedding grammars* are no more powerful than finite automata.

Counterpart devices for non-self-embedding grammars, for which the capability of recognizing recursive structures is limited by placing some restrictions on the size of the memory of the corresponding general model (*i.e.*, pushdown automata), are *constant-height pushdown automata*. More precisely, these devices are standard nondeterministic pushdown automata where the amount of available pushdown store is fixed. Hence, the number of their possible configurations is finite, thus implying that they are no more powerful than finite automata.

By contrast to models that make use of space or time restrictions, Hibbard introduced *d-scan limited automata* (or *d-limited automata*), that are obtained by limiting the writing capabilities of nondeterministic linear bounded automata allowing overwriting of each tape cell only the first d times that it is scanned, for some fixed $d \geq 0$ [Hib67]. Hibbard proved that, for each $d \geq 2$ these models characterize context-free languages. Further-

more, as shown by Wagner and Wechsung, when $d = 1$, that is, when these devices are allowed to overwrite the contents of each tape cell during the first visit only, 1-limited automata are equivalent to finite automata [WW86]. Moreover, it is a trivial observation that when $d = 0$, and hence no writings on the tape are allowed, 0-limited automata correspond to finite automata that can move their input head back and forth on the input tape, namely *two-way finite automata*.

In this work we shall focus on the models characterizing the bottom level of the Chomsky hierarchy, *i.e.*, the class of regular languages. We compare them from a descriptional complexity point of view, specifically, by studying their capability of representing the same class of languages in a more, or less, concise way. More precisely, *descriptional complexity* is a field of formal languages and automata theory whose goal is to find relationships between equivalent models in terms of their sizes, or, in other words, in the number of symbols that can be used to write down their descriptions.

A classical problem in this area is the investigation of the relationships between deterministic and nondeterministic devices. It is well known that *one-way deterministic* finite automata are sufficient for type 3 languages. By allowing nondeterministic transitions the computational power does not increase [RS59]. A natural question concerning models that share the same computational power is the comparison of their size. In this respect, even if deterministic and nondeterministic finite automata characterize the same class of languages, it is well known that, in the worst case, one-way deterministic automata can require exponentially many states with respect to equivalent nondeterministic automata. Hence, there is an exponential size gap from one-way nondeterministic to one-way deterministic automata.

Even by providing finite automata with the capability of moving the input head in both directions on the tape, their recognizing power does not increase [She59]. The models so obtained are *two-way* deterministic and nondeterministic finite automata.

At this point, one could ask *"Does the ability to scan the input in a two-way fashion in finite automata help in the elimination of the nondeterminism?"*. Differently from the one-way

case, the question about the size cost of the conversion of (one-way and two-way) nondeterministic finite automata into two-way deterministic finite automata, that was posed by Sakoda and Sipser in 1978, is still open. In their paper, Sakoda and Sipser conjectured that the elimination of the nondeterminism in the two-way case costs exponential. They also related this question to the well-known P $\stackrel{?}{=}$ NP problem [SS78].

The dissertation is organized as follows.

After a gather of preliminary mathematical concepts useful to understand the concepts presented in the thesis (Chapter 1), in Chapter 2 we define the models we deal with, and the measures of complexity we shall consider.

We start our study by investigating non-self-embedding grammars and their descriptional complexity [PP17]. In fact, in Chapter 3, we study the size costs of the conversion of non-self-embedding grammars into equivalent finite automata, by proving optimal bounds for the number of states of nondeterministic and deterministic automata equivalent to given non-self-embedding grammars. In particular, we show that each non-self-embedding grammar of size s can be converted into an equivalent nondeterministic automaton which has an exponential size in s and into an equivalent deterministic automaton which has a double exponential size in s. These costs are shown to be optimal.

We then continue the investigation about non-self-embedding grammars, by comparing them with constant-height pushdown automata and 1-limited automata [GPP18]. Since the double exponential gap in size from non-self-embedding grammars to deterministic finite automata, presented in Chapter 3, coincides with the gap in size from constant-height pushdown automata and 1-limited automata to deterministic finite automata, it is natural to ask how these models, obtained by posing some restrictions on grammars and recognizers characterizing context-free languages, are related to each other.

So, in Chapter 4 we prove that non-self-embedding grammars and constant-height pushdown automata are polynomially related in size, and we present a polynomial-size simulation by 1-limited automata. However, we prove that the converse transformation costs exponential. Finally, we give a different simulation that shows that also the conver-

sion of deterministic constant-height pushdown automata into deterministic 1-limited automata costs polynomial.

In Chapter 5 we focus on limited automata whose input alphabet is composed by just one symbol and we show that there exists an exponential gap between the size of limited automata accepting unary languages and the size of equivalent finite automata [PP19a]. Despite this gap, there are unary regular languages for which d-limited automata cannot be significantly smaller than finite automata, for any arbitrarily large d.

We also prove that from each unary context-free grammar $\mathcal{G}$ it is possible to obtain an equivalent 1-limited automaton whose description has a size that is polynomial in the size of $\mathcal{G}$.

In Chapter 6 we investigate, from a descriptional complexity point of view, the time complexity of 1-limited automata [GP19]. Though the model recognizes regular languages only, it may use quadratic time in the input length. We show that, with a polynomial increase in size and preserving determinism, each 1-limited automaton can be transformed into a linear-time equivalent one. We also obtain polynomial transformations into related models, including weight-reducing Hennie machines (*i.e.*, one-tape Turing machines syntactically forced to operate in linear-time), and we show exponential gaps for the converse transformations in the deterministic case.

The investigation about machines working in linear time is continued in Chapter 7, in which we prove that it is not decidable if a one-tape Turing machine works in linear time, even if it is deterministic and restricted to use only the portion of the tape which initially contains the input, unless the machine is weight-reducing [Gui+18].

By relating the study of these models to the above mentioned open question of Sakoda and Sipser, we study the costs of the conversion of nondeterministic finite automata into equivalent linear-time one-tape deterministic machines. We prove a polynomial blowup from two-way nondeterministic finite automata into equivalent weight-reducing one-tape deterministic machines (that work in linear time). The blowup remains polynomial if the tape in the resulting machines is restricted to the portion which initially contains

the input. However, in this case the resulting machines are not weight-reducing, unless the input alphabet is unary. Similar results are proved in the case the simulated nondeterministic automata are one-way.

In Chapter 8 we turn our attention to pushdown automata [PP19b]. In particular, we start our investigation by studying pushdown automata without any restriction on the input height and we show that it cannot be decided whether these devices accept using constant pushdown height, with respect to the input length, or not. Furthermore, in the case of acceptance in constant height, the height cannot be bounded by any recursive function in the size of the description of the machine. In contrast, in the restricted case of pushdown automata over a one-letter input alphabet, *i.e.*, unary pushdown automata, the above property becomes decidable. Moreover, if the height is bounded by a constant that does not depend on the input length, then it is at most exponential with respect to the size of the description of the pushdown automaton. This bound cannot be reduced. Finally, if a unary pushdown automaton uses non-constant height to accept, then the height should grow at least as the logarithm of the input length. This bound is optimal.

In conclusion, in Chapter 9 we briefly discuss some possible future research directions.

1 Preliminaries and Notation

In this chapter we recall some basic definitions and notations that will be used throughout this dissertation.

The standard notions from formal language and automata theory are presented by recalling the definitions in classical textbooks by Hopcroft and Ullman, and Shallit [HU79, Sha08].

1.1 Sets, Lists, and Tuples

A *set* is an unordered collection of distinct *elements*. Given a set S, if an element x is in S, it is denoted by $x \in S$, otherwise $x \notin S$.

If a set does not contain any element, it is called *empty set*, and it is denoted by $\emptyset$.

We denote by $\#S$ the number of elements of S, namely, its *cardinality*, that can be either *finite*, or *infinite*.

It is possible to describe a set S by enumerating all its elements within braces or by stating properties common to all the elements of S.

Example 1.1. The set composed by the first 13 prime numbers can be represented in exhaustive way by $S = \{ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41 \}$ or by stating a property characterizing it, by $S = \{ p \mid p < 42$ and p is a *prime number* $\}$, where the symbol $\mid$ is read "*such that*". It denotes the set of numbers x satisfying the following property:

- p is a number less than 42, and
- p is a prime number, *i.e.*, a number greater than 1 whose divisors are only 1 and itself. ■

Given two sets A and B it is also possible to define new sets by using the following *operations between sets*:

- The *intersection* of A and B: $A \cap B = \{ x \mid x \in A \text{ and } x \in B \}$.
- The *union* of A and B: $A \cup B = \{ x \mid x \in A \text{ or } x \in B \}$.
- The *difference* of A and B: $A \setminus B = \{ x \mid x \in A \text{ and } x \notin B \}$.

If all the elements of a set A are in a set B, then A is a *subset* of B, and it is denoted by $A \subseteq B$. A and B are *equal* (denoted by $A = B$), if they contain the same elements, or, in other words, if $A \subseteq B$ and $B \subseteq A$. A is a *proper subset* of B, denoted by $A \subset B$, if $A \subseteq B$ but A and B are not equal ($A \neq B$). If A and B do not have any element in common, *i.e.*, $A \cap B = \emptyset$, then we say that they are *disjoint*. The family of all the subsets of a set S is denoted by 2^S. Given a set $A \subseteq B$, the *complement* of A with respect to B is $A^c = B \setminus A$.

A collection $P = \{ S_i \}$ of nonempty sets is a *partition* of a set S if

- the sets are *pairwise disjoint*, *i.e.*, $S_i, S_j \in P$ and $i \neq j$ implies $S_i \cap S_j = \emptyset$, and
- their union is S.

The set of *natural numbers* is denoted by $\mathbb{N} = \{ 0, 1, 2, 3, \dots \}$.

A *list* is a collection in which the elements, that can be repeated (unlikely sets), are *ordered* by position.

A *tuple* is a list of fixed length. It is represented by enclosing its elements between parentheses. For example, $(0, 1, 2, 3, 4)$ is the tuple of length 5 (or 5-tuple) containing the first five natural numbers.

The *Cartesian product* (or simply *product*) of n sets $A_1, A_2, \dots, A_n$, denoted by $A_1 \times A_2 \times \cdots \times A_n$, is the set of tuples of length n (namely, *n-tuples*) such that the i-th element

of each n-tuple is an element of A_i, for $i = 1, \ldots, n$, formally

$$A_1 \times A_2 \times \cdots \times A_n = \{ (a_1, a_2, \ldots, a_n) \mid a_i \in A_i,\ i = 1, \ldots, n \}.$$

The *n-th power* of a set A, is the Cartesian product of A repeated n times, in symbols

$$A^n = \underbrace{A \times A \times \cdots \times A}_{n \text{ times}}.$$

1.2 Graphs and Trees

A *(directed) graph* is a pair $G = \langle V, E \rangle$, in which V is the finite set of *nodes* (or *vertices*) and E is the finite set of *edges* connecting pairs of nodes. More formally, $E : V \times V$ is a binary relation on V, and given two nodes $u, v \in V$, the edge connecting u to v is denoted by the pair (u, v). A graph is *undirected* if, for each edge (u, v) in E, there also exists the edge (v, u) in E.

A *path of length k* from a vertex u to a vertex v in a graph $G = \langle V, E \rangle$ is a sequence $v_0, v_1, v_2, \ldots, v_k$ of vertices such that $u = v_0$, $v = v_k$, and $(v_{i-1}, v_i) \in E$, for $i = 1, 2, \ldots, k$. The *length of the path* is the number of edges in it. The path contains the vertices $v_0, \ldots, v_k$ and the edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$. If there is a path from u to v, we say that v is *reachable* from u.

A path $v_0, v_1, \ldots, v_k$ forms a *cycle* if $v_0 = v_k$ and the path contains at least one edge. A cycle is said to be *simple* if the only repeated vertices are v_0 and v_k. A graph with no cycles is *acyclic*.

A *strongly connected component* (SCC, for short) of a directed graph $G = \langle V, E \rangle$ is a maximal subset V' of V such that for each pair of vertices $u, v \in V'$, G contains a path from u to v. If $V' = V$, *i.e.*, all the nodes of the graph form a unique SCC, then G is said to be *strongly connected*. An SCC is *trivial* if it does not contain any loop, namely, it is a single vertex v without the edge (v, v). Otherwise, it is said to be *nontrivial*.

A *tree* is an undirected connected acyclic graph. A tree in which one of the nodes (*root*) is distinguished from the others is a *rooted tree*. All the trees that we shall consider in this thesis are rooted, and on them we use standard terminology from graph theory.

1.3 Basic Notions of Automata Theory

An *alphabet* is a finite, nonempty set of *symbols* or *letters*. As convention, we will often use the symbols Σ and Γ for alphabets.

A *string* or *word* w over some alphabet Σ is a finite sequence of symbols chosen from Σ. We denote by $|w|$ the *length* of w, by $w[i]$ the i-th symbol of w, for $i = 1, \ldots, |w|$, and by ε the *empty string*, that is the string composed by no letters. Notice that ε is the only string whose length is 0. For any $h \geq 0$, Σ^h denotes the set of strings of length h over the alphabet Σ.

The set of all words over Σ is denoted Σ^*, while, if the empty string is excluded from such a set, it is denoted by Σ^+.

Let x and y be strings. Then xy is the string obtained from the *concatenation* of x and y, that is the string composed by a copy of x followed by a copy of y. Given a string w, $w = a_1 a_2 \cdots a_n$, let us denote by w^R the *reverse* of w, *i.e.* the string $a_n \cdots a_2 a_1$, and by w^n the string composed by the concatenation of n copies of w. If $n = 0$, then $w^0 = \varepsilon$. Moreover w^* denotes the set of words obtained by concatenating w arbitrarily many times (commonly known as *Kleene star* or simply *star* operation): $w^* = \{ \varepsilon, w, w^2, w^3, \ldots \}$.

Given an alphabet Σ, the set L of strings chosen from Σ^* is a *language* over Σ. The *empty language* $\emptyset$ is the language composed by no words. Notice that it is different from the language composed by the empty word only. A *unary language* is defined over a one-letter alphabet.

Since languages are set of words, classical operations on sets like union, intersection and complementation are extended to languages in a natural way.

The *product* or *concatenation* of two languages L_1 and L_2 is the language

$$L_1 L_2 = \{ uv \mid u \in L_1 \text{ and } v \in L_2 \}.$$

Given a language L, we denote by L^i, $i \geq 1$, the concatenation of L for i times, formally $L^i = L^{i-1} L$, with $L^0 = \{ \varepsilon \}$. Moreover, L^* denotes the language obtained by concatenate the strings in L arbitrarily many times, *i.e.*, $L^* = L^0 \cup L^1 \cup L^2 \cup \ldots$, and $L^+ = LL^*$.

1.3.1 Grammars

A *(formal) grammar* is a tuple $\mathcal{G} = \langle V, \Sigma, P, S \rangle$, where V is a finite set of *variables* (or *nonterminal symbols*, or just *nonterminals*), T is a finite set of *terminal symbols* (or *terminals*) that can occur in the strings of the language being defined, $S \in V$ is the *start symbol* (or *initial symbol*), representing the language being defined, and P is the finite set of *production rules* (or *productions*) that represent the definition of a language. Each production has the form

$$\alpha \to \beta$$

in which

- α is a nonempty string of terminals and variables, in which at least one nonterminal occur, called *left-hand side* (or *head*) of the production,
- $\to$ is the production symbol, and
- β is a (possibly empty) string of terminals and variables, called *right-hand side* (or *body*) of the production. It represents one way of replacing the head during the *derivation*, that is the process of generation of a string starting from the initial symbol and iteratively applying production rules.

Given two variables $A, B \in V$, productions of the form $A \to B$ and $A \to \varepsilon$ are called *unit productions* and *ε-productions*, respectively.

Starting from the initial symbol S, it is expanded by using one of its productions (*i.e.*, using a production whose head is S). The string s obtained is further expanded by picking one production $\alpha \to \beta$ whose head occurs in s, and by replacing α with β. This is repeated until s consists entirely of terminal symbols.

The process of derivation is indicated by using the relation $\Rightarrow$. Let $\alpha'\alpha\alpha'' \in (V \cup T)^*$ be a string of terminal and variables and $\alpha \to \beta \in P$ be a production of $\mathcal{G}$. Then we denote $\alpha'\alpha\alpha'' \underset{\mathcal{G}}{\Rightarrow} \alpha'\beta\alpha''$ for one step of derivation of $\mathcal{G}$ (or simply $\alpha'\alpha\alpha'' \Rightarrow \alpha'\beta\alpha''$ if $\mathcal{G}$ is clear from the context). We denote $\overset{\star}{\Rightarrow}$ the extension of $\Rightarrow$ representing zero or more steps of derivation.

Derivations from the start symbol produce strings that are called *sentential forms*. Thus, any string $\alpha \in (V \cup T)^*$ such that $S \overset{\star}{\Rightarrow} \alpha$ is a sentential form.

The language generated by $\mathcal{G}$ is denoted by $L(\mathcal{G})$, and it contains all the sentential forms composed only by terminal symbols. In symbols $L(\mathcal{G}) = \{w \mid S \stackrel{\star}{\Rightarrow} w, w \in T^*\}$.

It is possible to represent the derivations of a grammar $\mathcal{G}$ as a tree, namely the *derivation* (or *parse*) *tree*, in which:

- Each node is labeled by a variable in V.
- Each leaf is labeled by either a variable, a terminal, or ε.
- If an interior node is labeled A and its children are labeled $X_1, X_2, \ldots, X_k$, respectively, from the left, then $A \to X_1 X_2 \cdots X_K$ is a production in P.

2

Computation Models and Descriptional Complexity

In this chapter we shall introduce the models studied in this thesis and the complexity measures we consider.

We start by recalling the Chomsky hierarchy (Section 2.1), a hierarchy of class of languages, characterized by formal grammars that, for each level of the hierarchy, have stronger and stronger restricting conditions. Each class of the Chomsky hierarchy is also characterized in terms of acceptors, *i.e.*, computational models that, given a word w in input, are able to determine the membership of w to a language. Therefore, in Section 2.2 we define grammars and acceptors we are going to deal with, by posing some restrictions on more general models.

Besides the references given in the following sections, more details on grammars and devices can be found in classical text-books (*e.g.*, [HU79, Sha08]).

Finally in Section 2.3, we introduce the measures of complexity of these models and we present the problems we intend to investigate and some classical examples in the area of descriptional complexity, according to the reference survey by Goldstine et al. [Gol+02].

2.1 Restrictions of Formal Grammars: The Chomsky Hierarchy

In one of his pioneer papers, Chomsky defined the following classes of languages, that are generated by grammars that were studied as potential models for the generation of natural languages [Cho59a].

Type 0 languages. The largest family of the Chomsky hierarchy is the class of languages recognized by Turing machines (in both deterministic and nondeterministic versions), even in the variant with one tape only. This family is generated by *unrestricted* — or *type 0* — grammars, whose productions are of the form $\alpha \to \beta$, where α and β are arbitrary strings of grammar symbols, with $\alpha \neq \varepsilon$. In other words, these are general grammars that do not have any kind of restriction.

It is possible to recognize a type 0 language by using a Turing machine, (that, as explained in the following section, is a general model of computation), that nondeterministically guesses the "backward" derivation used to obtain the word given in input. This procedure is implemented as follows. At each step of computation, the machine guesses the production $\alpha \to \beta$ that was applied at the previous step of the derivation to obtain the current string and replaces the portion of the word corresponding to β with α. This process is iterated until, simulating the recursive inference, the initial symbol of the grammar is found and it is the only remaining symbol of the string.

Type 1 languages. The weakest restriction on grammars posed by Chomsky was that characterizing the class of *context-sensitive* (CSL, for short) — or *type 1* — languages. In context-sensitive grammars, each production has the form $\alpha_1 A \alpha_2 \to \alpha_1 \beta \alpha_2$, in which A is a variable and α_1, α_2, and $\beta \neq \varepsilon$ are strings of grammar symbols. Productions of this type mean that it is possible to replace the variable A with the nonempty string β when A is in the "context" given by α_1 and α_2, *i.e.*, between α_1 and α_2. The

only exception is a production $S \to \varepsilon$, provided that the variable S does not appear on the right-hand side of any production.

Notice that, since $\beta \neq \varepsilon$, the right-hand side of each production is never shorter than the left-hand side.

Also in this case, it is possible to design a Turing machine that performs a "backward" guessing of the productions used in the derivation process. Differently from the type 0 case, since the right-hand sides of the productions are shorter than the left-hand sides, the machine should obtain a shorter and shorter string for each step of the "backward" derivation, so it uses an amount of space that is limited by the length of the input. When a Turing machine uses just the portion of the tape that initially contains the input, as in this case, the resulting model is called *linear-bounded automata*. So, linear-bounded automata are the device counterpart characterizing this class of languages.

Type 2 languages. *Context-free* grammars, that characterize this class of languages, namely *context-free languages* (CFL, for short), have the capability of generating recursive structures. They are obtained by restricting context-sensitive grammars. In particular, each production has the form $\alpha_1 A \alpha_2 \to \alpha_1 \beta \alpha_2$, with $\alpha_1 = \alpha_2 = \varepsilon$. It is possible to notice that, in this case, it is always possible to apply the production from the variable A, independently from the "context" surrounding it.

The ability of recognizing recursive structures in languages is implemented by using the pushdown store of *pushdown automata*.

Type 3 languages. Finally, the smallest class of the hierarchy is the one of the *regular languages* (REG, for short). The productions of the grammars generating regular languages, namely *regular* — or *type 3* — grammars, have the form $A \to aB$ or $A \to a$, where A and B are variables and a is a terminal symbol, or $S \to \varepsilon$, provided that the variable S does not appear on the right-hand side of any production.

Typical acceptors for these languages are *finite automata*, that are simple machines

that are able to represent only a finite set of possible configurations.

2.2 General Devices and Restrictions Characterizing Regular Languages

In this thesis we shall analyze, from a descriptional complexity point of view, variants of classical formal computational models that characterize type 3 languages, obtained by posing some restrictions on the computational resources of more general devices. This section is devoted to the formal definition of these models.

2.2.1 Turing machines

Turing machines, introduced by Turing in 1936, were proposed as a general model of computation, formalizing the concept of procedure [Göd31] and, hence, the intuitive notion of algorithm [Kle52]. Such a principle is known as *Church's Thesis*. As seen in the previous section, this device was proven to be equivalent to unrestricted grammars (see, *e.g.* [Cho59a]) in both the deterministic and nondeterministic versions.

In this thesis we shall consider *one-tape Turing machine* (TM), that are devices consisting of a *finite control*, composed by the set of states the machine can be into during the computation, an infinite *tape*, divided into *cells*, each one containing one letter of the input, and a *head* that is used to read and write the tape contents. At the beginning of the computation, the *input* of the Turing machines, a finite-length string of symbols chosen from the *input alphabet*, is placed on a segment of an infinite tape, called *initial segment*, surrounded on the left and on the right by infinite sequences of special symbols $\not\emptyset$ called *blank*, and the head is at the leftmost cell containing the input.

At each step of computation, the machine, depending on its configuration — given by the current state and the symbol scanned by the head — and according to the set of its *instructions*, defined through a *transition function* that can be seen as the program that describes the behavior of the machine, reaches the next configuration, by updating its

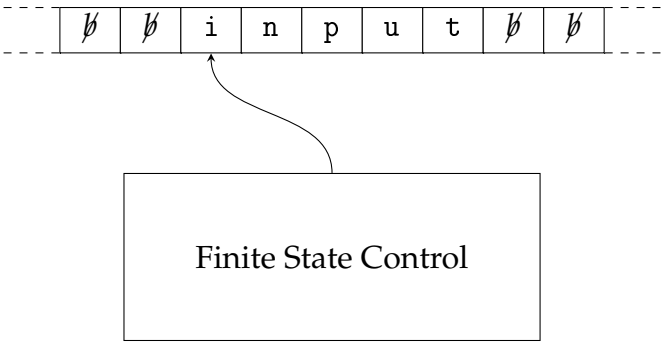

Figure 2.1: The representation of a Turing machine. It is composed by a finite state control and a head used to read and write the cells of the tape.

internal state, overwriting the symbol currently scanned with a symbol from a *working alphabet*, and moving the tape head to left, to right, or keeping it in place.

Formally,

Definition 2.1. *A* Turing machine *is a tuple* $\langle Q, \Sigma, \Gamma, \delta, q_I, F \rangle$ *where* Q *is a finite* set of states, Σ *is a finite* input alphabet, Γ *is a finite* working alphabet *including* Σ, $q_I \in Q$ *is the* initial state, $F \subseteq Q$ *is the* set of final states, *and* $\delta : Q \times \Gamma \to 2^{Q \times \Gamma \times \{-1,0,+1\}}$ *is the* transition function.

When $\#\delta(p, \sigma) \leq 1$ *for each* $p \in Q$ *and* $\sigma \in \Gamma$, *the Turing machine is* deterministic (DTM, *for short). In this case, the* deterministic transition function *is* $\delta : Q \times \Gamma \to Q \times \Gamma \times \{-1, 0, +1\}$.

Since we shall focus on DTMs let us describe more in detail the meaning of the transition function of these devices. In one step, depending on its current state p and on the symbol σ read by the head, a DTM changes its state to q, overwrites the corresponding tape cell with τ and moves the head one cell to the left, to the right, or leaves it on the same cell if $\delta(p, \sigma) = (q, \tau, d)$, according to the fact that $d = -1$, $d = +1$, or $d = 0$.

The computation of a Turing machine $\mathcal{A}$ over an input w starts in the initial state with the head scanning the leftmost symbol of w. This is the *initial configuration* of $\mathcal{A}$. The input is *accepted* if the machine eventually *halts* in a final state, namely, if it reaches a final state from which, given the current configuration, no move is possible.

A *configuration* is represented as a string zpz', meaning that p is the current state,

$zz' \in \Pi^*$ is the content of the tape (where Π denotes the set of symbols that can appear on the tape of the model) and the head is scanning the first symbol of z'. The transition relation between configurations is denoted by $\vdash$, and its reflexive-transitive closure by $\vdash^*$. We also represent *partial configurations* as upv, where p is the current state and uv is a factor of the tape content.

In order to understand how a Turing machine works, let us warm up with a preliminary example.

Example 2.1. Let us consider the following language:

$$L = \{\, ww \mid w \in \{\, a, b \,\}^* \,\}$$

composed by the square of a word w on the alphabet $\{\, a, b \,\}$.

We are going to define a Turing machine $\mathcal{T}$ accepting L and to explain how it works.

Roughly, at the beginning of the computation the machine guesses which is the position c of the cell containing the first symbol of the second occurrence of w. So, it compares the symbol in position 1 with the one in position c, and, if they are equal, $\mathcal{T}$ overwrites them with a special marker. Then, it compares the symbols in position 2 and $c + 1$, overwriting them if they are equal, and so on, until reaching the end of the string. If, by repeating these steps, the end of the input is reached and all the input symbols have been overwritten, then the machine accepts.

Formally, let $\mathcal{T} = \langle Q, \Sigma, \Gamma, \delta, q_I, F \rangle$, where $Q = \{\, q_I, q_1, q_2, \ldots, q_9, q_F, q_T \,\}$, $\Sigma = \{\, a, b \,\}$, $\Gamma = \{\, \mathsf{X}, \not\emptyset \,\} \cup \Sigma$, q_I is the initial state, and the set of final states is $F = \{\, q_I, q_F \,\}$. The transition function δ is defined as follows (undefined transitions are not listed):

i. $\delta(q_I, a) = \{\, (q_1, \mathsf{X}, +1) \,\}$

ii. $\delta(q_I, b) = \{\, (q_2, \mathsf{X}, +1) \,\}$

iii. $\delta(q_1, a) = \{\, (q_1, a, +1), (q_3, \mathsf{X}, -1) \,\}$

iv. $\delta(q_1, b) = \{\, (q_1, b, +1) \,\}$

v. $\delta(q_2, b) = \{ (q_2, b, +1), (q_3, \mathsf{X}, -1) \}$

vi. $\delta(q_2, a) = \{ (q_2, a, +1) \}$

vii. $\delta(q_3, \sigma) = \{ (q_3, \sigma, -1) \} = \delta(q_9, \sigma)$, for $\sigma \in \Sigma$

viii. $\delta(q_3, \mathsf{X}) = \{ (q_4, \mathsf{X}, +1) \}$

ix. $\delta(q_4, a) = \{ (q_5, \mathsf{X}, +1) \}$

x. $\delta(q_4, b) = \{ (q_6, \mathsf{X}, +1) \}$

xi. $\delta(q_5, \sigma) = \{ (q_5, \sigma, +1) \}$, for $\sigma \in \Sigma$

xii. $\delta(q_5, \mathsf{X}) = \delta(q_7, \mathsf{X}) = \{ (q_7, \mathsf{X}, +1) \}$

xiii. $\delta(q_6, \sigma) = \{ (q_6, \sigma, +1) \}$, for $\sigma \in \Sigma$

xiv. $\delta(q_6, \mathsf{X}) = \delta(q_8, \mathsf{X}) = \{ (q_8, \mathsf{X}, +1) \}$

xv. $\delta(q_7, a) = \delta(q_8, b) = \{ (q_9, \mathsf{X}, -1) \} = \delta(q_9, \mathsf{X})$

xvi. $\delta(q_9, \not\flat) = \{ (q_F, \not\flat, +1) \}$

xvii. $\delta(q_F, \mathsf{X}) = \{ (q_F, \mathsf{X}, +1) \} = \delta(q_4, \mathsf{X})$

xviii. $\delta(q_F, \sigma) = \{ (q_T, \sigma, +1) \}$, for $\sigma \in \Sigma$

The machine starts in the initial state q_I. At the beginning of the computation, it stores the first symbol of the input word (and hence of the word w) in its finite control and overwrites it with the symbol X (Transitions i. and ii.). Then, it guesses which is the cell of the tape containing the first letter of the second occurrence of w and, after reaching it, $\mathcal{T}$ overwrites such a cell with X if its contents corresponds to the letter stored in the finite control (Transitions iii., iv., v., and vi.). At this point, the first letter of both the occurrences of w is marked with X. So, the machine repeats the following steps:

1. it moves the head backward until reaching the leftmost unmarked symbol $\sigma \in \Sigma$ of the first occurrence of w (Transitions vii. and viii.);

2. it stores σ in the finite control and overwrites σ with X (Transitions ix. and x.);

3. it moves the head to the right until reaching the leftmost unmarked symbol of the second occurrence of w (Transitions xi., xii., xiii., and xiv.);

4. it compares the symbol scanned by the head with the one stored in the finite control and, if they correspond, the machine continues the execution by starting another iteration of this procedure from Step 1 (Transitions xv.).

Thus, $\mathcal{T}$ exits this loop

- either when one of the symbols of the second occurrence of w does not correspond to the symbol stored in the finite control,

- or when, by moving the head backward while executing Step 1, a blank symbol $\not\emptyset$ is reached (Transition xvi.).

In the former case, $\mathcal{T}$ halts and rejects (undefined transitions). While, in the latter case, the device performs a last scan of the tape to check if no symbols of the input word x are left, because at this point all of them should have been overwritten if x is in the language (Transitions xvii.). In this case, the $\mathcal{T}$ halts and accepts. Otherwise, as soon as one input symbol is detected, the machine enters a nonaccepting state and halts (Transition xviii.).

■

Let us now consider a trickier example, which will be utilized later.

Example 2.2 ([Sha08])**.** Consider the deterministic Turing machine $\mathcal{BB}_3$ represented in the *transition graph* in Figure 2.2, in which each state is represented by a node, and each transition $\delta(p, a) = (q, X, d)$ is represented by an edge from the state p to q labeled by $a/X, d$.

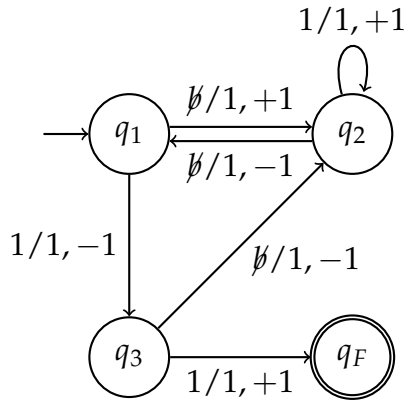

Figure 2.2: An example of deterministic Turing machine for the busy beaver problem.

The initial state is denoted by an arrow entering it, and the final state is represented by a double circle.[1]

Such a machine, starting with the tape containing just an infinite sequence of blank symbols, ends the computation with a string on the tape in which the number of 1's is the maximum that is possible to obtain with 3 states plus an accepting one from which no outgoing transitions are defined. In fact, it halts after 13 moves with 6 consecutive 1's on the tape. Such a machine is a 3-state *busy beaver* (we do not count the halting state in the total number of states). ■

In general, for $n > 0$, a *busy beaver* $\mathcal{BB}_n$ is a deterministic Turing machine with a set of n states Q_n and tape alphabet $\Gamma = \{ 1, \not b \}$, that starting with an empty tape, ends the computation with a string on the tape in which the number of 1's, denoted as $\Sigma(n)$, is maximum. We call $\Sigma : \mathbb{N} \to \mathbb{N}$ the *busy beaver function*.

Even if for small arguments of the busy beaver function it is possible to compute $\Sigma(n)$ — as in our case, in which $\Sigma(3) = 6$ — Radó in 1962 proved that $\Sigma(n)$ grows asymptotically faster than any recursive function:

Theorem 2.1 ([Rad62]). $\Sigma(n)$ *cannot be bounded by any recursive function.*

[1] In the following, transitions graphs will be also used to represent the other devices we are going to study. In those cases, the edges will be labeled in accordance with the transition function of the model under consideration.

2.2.1.1 Restrictions of Turing machines

Since Turing machines are a general model of computation equivalent to the concept of algorithm (according to Church's Thesis), this model and its variants have been widely studied in the literature. For example, in the area of structural complexity, classical complexity classes such as P, NP, LOGSPACE, *etc.* are defined by introducing a limit on the amount of resources, specifically time or space, at disposal of the model. Usually, such limitations reduce the expressive power.

In this work we shall consider Turing machines operating under some limitations on their computational resources, thus obtaining less powerful devices that characterize smaller families of languages. In particular, the following restrictions of DTMs will be considered.

Bounded machines. We say that a device is *bounded*, if each input string is surrounded by two special symbols called the left and the right endmarkers, $\triangleright$ and $\triangleleft$ respectively, and for each transition $\delta(p, \sigma) = (q, \tau, d)$, $\sigma = \triangleright$ (resp., $\sigma = \triangleleft$) implies $\tau = \sigma$ and $d = +1$ (resp., $d = -1$). Hence, the machine is restricted to use only the initial segment (plus the endmarkers), that cannot be left by the head, *i.e.*, it cannot be moved to the left of the cell containing $\triangleright$ and to the right of the cell containing $\triangleleft$ and, moreover, these endmarkers cannot be modified.

Linear-bounded automata are bounded TMs [Kur64], while we call *bounded* DTMs their deterministic restriction.

Weight-reducing Turing machines. Roughly speaking, in *weight-reducing* Turing machines each overwriting is decreasing with respect to some fixed order on the working alphabet. As a consequence, after overwriting a cell with a minimal symbol, such a machine cannot visit the cell again.

Formally, a Turing machine is weight-reducing (*wr* TM), if there exists an order $<$ on Γ such that $\delta(p, \sigma) \ni (q, \tau, d)$ implies $\tau < \sigma$. By this condition, in a wrTM the number of visits to each tape cell is bounded by a constant. However, one wrTM

could have non-halting computations which, hence, necessarily visit infinitely many tape cells.

Deterministic weight-reducing Turing machines are denoted wrDTMs.

Linear-time Turing machines. A Turing machine is said to be *linear-time* if, over each input w, its computation halts within $O(|w|)$ steps.

Hennie machines. A *Hennie machine* (HM) is a linear-time Turing machine which is, furthermore, bounded.

Deterministic Hennie machines are denoted DHMs.

Weight-Reducing Hennie machines. By combining previous conditions, *weight-reducing Hennie machines* are defined, in both nondeterministic (*wr* HMs) and deterministic (*wr* DHMs) versions.

Observe that each bounded wrDTM can execute a number of steps which is at most linear in the length of the input. Hence, bounded wrDTMs are necessarily Hennie machines.

Further restrictions deriving from finer analyses of computational resources of Turing machines will be considered through this thesis. We shall introduce these models by expressing the main differences with the general model of Turing machine just defined.

2.2.1.2 A note on common notations for restrictions of Turing machines

When considering restrictions of Turing machines some definitions could slightly differ from the general model.

In particular, for machines that are allowed to use only the portion of the tape that initially contains the input, at the beginning of the computation the input word w is stored on the tape surrounded by the two end-markers, the left end-marker being at the position zero. Hence, the right end-marker is on the cell in position $|w| + 1$. The head of the machine is on the cell 1.

The input is accepted if and only if there is a computation path which starts from the initial configuration and ends in a final state *after violating the right end-marker*, *i.e.*, with the head which leaves the tape by moving to the right of the right end-marker.

In this case, configurations are represented by strings of the form zpz', where p is the current state, $zz' \in \triangleright\Pi^*\triangleleft$ is the content of the tape, for some alphabet Π, and the head is scanning the first symbol of z'. Notice that, in case $|z'| = 0$, the machine has reached the end of the computation.

In the cases in which we want to emphasize computations involving only some portion of the tape, we shall use *partial configurations*. Partial configurations are represented as strings of the form upv, where p is the current state and $uv \in \{\varepsilon, \triangleright\}\Pi^*\{\varepsilon, \triangleleft\}$ is a factor of the tape content. The relations $\vdash$ and $\vdash^*$ on configurations extend onto partial configurations.

Instead, for *one-way machines*, *i.e.*, machines whose head never moves to the left, the input w is accepted if and only if there is a computation path which starts from the initial configuration and ends in a final state *after reading the last symbol of the* w.

The language accepted by an acceptor $\mathcal{A}$ is denoted by $L(\mathcal{A})$ and is composed by all the input words accepted by $\mathcal{A}$. If $L(\mathcal{A})$ is a unary language, or, in other words, the input alphabet is composed by one symbol only, then $\mathcal{A}$ is said to be *unary* as well.

A transition that does not depend on the input symbol scanned by the head, *i.e.*, it is performed without reading any input symbol, is called ε-transition (or ε-move).

2.2.2 Limited automata

By contrast to the standard restrictions on time and space, Hibbard in 1967 introduced *limited automata* (originally called *scan-limited automata*), that are obtained by posing a finer restriction of the capabilities of Turing machines. In particular, *d-limited automata* are linear-bounded automata that, after the first d visits to a cell, for a fixed integer $d \geq 0$, are not allowed to overwrite its contents anymore. Nevertheless, the cell may be visited again and the information stored therein read arbitrarily many more times, but its contents is

frozen for the remainder of the computation [Hib67].

Formally, limited automata are defined as follows:

Definition 2.2. *Given an integer $d \geq 0$, a* d-limited automaton *(d-LA, for short) is a tuple $\mathcal{A} = \langle Q, \Sigma, \Gamma, \delta, q_I, F \rangle$, where Q is a finite* set of states, *Σ is a finite* input alphabet, *Γ is a finite* working alphabet *such that $\Sigma \cup \{ \rhd, \lhd \} \subseteq \Gamma$, where $\rhd, \lhd \notin \Sigma$ are the* left *and the* right end-markers, *respectively, $\delta : Q \times \Gamma_{\rhd\lhd} \to 2^{Q \times \Gamma_{\rhd\lhd} \times \{ -1,0,+1 \}}$ is the* nondeterministic transition function, *where $\Gamma_{\rhd\lhd}$ denotes the set $\Gamma \cup \{ \rhd, \lhd \}$, $q_I \in Q$ is the* initial state, *and $F \subseteq Q$ is the* set of final states.

In one move, according to δ and to the current state, $\mathcal{A}$ reads a symbol from the tape, changes its state, replaces the symbol just read from the tape by a new symbol, and moves its head, as happens for Turing machines. However, there are the following restrictions:

- *Replacing symbols is allowed to modify the contents of each cell only during the first d visits, with the exception of the cells containing the end-markers, which are never modified.*
- *The end-marker symbols cannot be used to replace the contents of any of the cells which initially contains the input. (With the previous condition, this implies that if $(q, X, m) \in \delta(p, a)$ and either $X \in \{ \rhd, \lhd \}$ or $a \in \{ \rhd, \lhd \}$, then $X = a$.)*
- *The head cannot violate the end-markers,* i.e., *it cannot move to the left of the left end-marker or to the right of the right end-marker, except at the end of computation, to accept the input, as explained in Section 2.2.1.2.*

An automaton $\mathcal{A}$ is said to be limited *if it is d-limited for some $d \geq 0$.*

As for Turing machines, $\mathcal{A}$ is said to be *deterministic* (deterministic d-LA, for short) whenever $\#\delta(q, a) \leq 1$, for any $q \in Q$ and $a \in \Gamma$.

2.2.2.1 A classical example: recognizing Dyck languages

In order to show how this model works, we shall illustrate a classical example of execution for accepting the set of words representing well balanced sequences of brackets, namely the *Dyck languages*.

Definition 2.3. *A* bracket alphabet

$$\Omega_k = \{\ (_1, (_2, \ldots, (_k,)_1,)_2, \ldots,)_k \}$$

is a finite set containing an even number of symbols, say $2k$*, with* $k > 0$*, where the first k symbols are interpreted as* left brackets *of k different types, while the remaining symbols are interpreted as the corresponding* right brackets*. The* Dyck language $\mathcal{D}_{\Omega_k}$ *over* Ω_k *is the set of all sequences of balanced brackets from* Ω_k*.*

Example 2.3 ([Pig19])**.** Let us consider the *Dyck language* $\mathcal{D}_{\Omega_k}$ over the alphabet Ω_k of k types of brackets.

To recognize the strings in $\mathcal{D}_{\Omega_k}$, a limited automaton can use the following strategy. It starts by scanning the input tape from the left to the right until reaching the leftmost cell containing a closing bracket. The corresponding opening one is necessarily the last bracket before it, that must be of the same type. Such a bracket is located by changing the direction of the head moving it backward along the tape. The two corresponding brackets are then removed and overwritten with a special symbol. At this point, the same procedure, consisting in

- locating the first (leftmost) closing bracket,
- checking if the last bracket before it is of the same type,
- and overwriting these two brackets,

can be iterated.

When no more closing brackets are left on the tape, opening bracket can be left neither. In this case the original word was balanced. Otherwise, in the following cases the sequence is not balanced:

- At the end of the procedure the sequence left on the tape contains some opening bracket;
- After locating a closing bracket no opening one before it is found;
- After locating a closing bracket the last opening one, found before it, is of a different type.

Let us focus on the movements of the head of a machine implementing the described procedure. The computation of the device starts with the head scanning the cell containing the first input symbol. So, each input cell is reached for the first time while moving the head from left to right. Moreover, the following holds:

- A cell containing a closing bracket is overwritten when the head visits it for the first time. After that, the head is moved backward to look for an opening bracket;
- A cell containing an opening bracket is overwritten when the head reaches it for the second time, while the device is scanning the tape from the right to the left;
- After the cell is overwritten, it could be visited further many times, but it is not overwritten anymore.

Hence, a device implementing such a procedure, that overwrites each cell only in the *first two visits* and, after that, it never modifies their contents, is a 2-*limited automaton*. ■

The device just described accepts the Dyck languages. It is easy to notice that, with a slight modification to the given procedure, it is possible to recognize an extended version of a Dyck language $\mathcal{D}_{\Omega_k}$, that is obtained by pumping the strings of $\mathcal{D}_{\Omega_k}$ with some extra symbols that, during the recognizing procedure, will be just ignored. Formally,

Definition 2.4. *An* extended bracket alphabet Ω *is a nonempty finite set which is the union of two, possibly empty, sets* Ω_k *and* Ω_n*, where* Ω_k*, if not empty, is a bracket alphabet, and* Ω_n *is a set of* neutral *symbols. The* extended Dyck language $\widehat{\mathcal{D}}_\Omega$ *over* Ω *is the set of all the strings that can be obtained by arbitrarily inserting symbols from* Ω_n *in strings of* $\mathcal{D}_{\Omega_k}$*.*

Example 2.4. Let $\Omega_k = \{\ (, [\ ,\ ),\ ]\ \}$, $\Omega_n = \{\ |\ \}$, and $\Omega = \Omega_k \cup \Omega_n$. Then $([[\,]\,])[\,] \in \mathcal{D}_{\Omega_k} \subset \widehat{\mathcal{D}}_\Omega$. ■

Dyck languages are important in the investigation of context-free languages, because they capture the recursive structure of any context-free language. This fact is formalized by the famous Chomsky-Schützenberger representation theorem for context-free languages [CS63], that we present here in a nonerasing variant, proved by Okhotin [Okh12]:

Theorem 2.2. *A language $L \subseteq \Sigma^*$ is context free if and only if there exist an extended bracket alphabet Ω, a regular language $R \subseteq \Omega^*$, and a letter-to-letter homomorphism $h : \Omega \to \Sigma$ such that $L = h(\widehat{\mathcal{D}}_\Omega \cap R)$.*

Theorem 2.2 suggests a way to build, for any context-free language L, a recognizer which is the combination of the following devices:

- A one-way nondeterministic machine $\mathcal{T}$ that, on each input $w \in \Sigma^*$, produces a string $z \in h^{-1}(w)$;
- A machine $\mathcal{A}_D$ recognizing the extended Dyck language $\widehat{\mathcal{D}}_\Omega$;
- A one-way finite automaton $\mathcal{A}_R$ accepting the regular language R.

As device $\mathcal{A}_D$ we can use a 2-limited automaton implementing the procedure outlined in Example 2.3, with the modification that allows to accept the extended Dyck language $\widehat{\mathcal{D}}_\Omega$. As a conclusion, each context-free language is accepted by a 2-limited automaton. (For further details, see [Pig15, PP14].)

As proved by Hibbard, the converse also holds. Furthermore, by allowing a larger — but still constant — number of visits during which it is possible to rewrite each tape cell, the computational power does not change:

Theorem 2.3 ([Hib67])**.** *For each $d \geq 2$, the class of languages accepted by d-limited automata coincides with the class of context-free languages.*

Hibbard furthermore showed the existence of an infinite hierarchy of deterministic d-limited automata, whose first level (*i.e.*, corresponding to deterministic 2-limited automata) has been later proved to coincide with the class of *deterministic context-free languages* [PP15]. (See [KPW18] and references therein for further connections between limited automata and context-free languages.)

2.2.2.2 1-limited automata

Let us now turn our attention on the case $d = 1$. In 1986, Wagner and Wechsung proved that 1-limited automata characterize the class of regular languages [WW86, Thm. 12.1].

Remark 2.1. *A* 1-limited automaton $\mathcal{A}$ *(*1-LA*, for short) is a limited automaton in which, for each transition* $(q, \gamma, d) \in \delta(p, \sigma)$*, we have* $\gamma \in \Gamma_{\triangleright\triangleleft} \setminus \Sigma$ *and if* $\sigma \in \Gamma_{\triangleright\triangleleft} \setminus \Sigma$ *then* $\gamma = \sigma$*.*

In one move, according to δ and to the current state, $\mathcal{A}$ reads a symbol from the tape, changes its state, replaces the symbol just read by a new symbol, and moves its head one position backward or forward or keeps it in place. However, replacing symbols is subject to some restrictions, which, essentially, allow to modify the content of a cell during the *first visit only*. Technically, symbols from Σ shall be replaced by symbols from $\Gamma \setminus \Sigma$, while symbols from $\Gamma_{\triangleright\triangleleft} \setminus \Sigma$ are never overwritten. In particular, at any time, both special symbols $\triangleright$ and $\triangleleft$ occur exactly once on the tape and exactly at the respective left and right boundaries. Acceptance for 1-LAs, as well as *deterministic* 1-LAs, is defined exactly as for bounded machines (*cf.*, Section 2.2.1.2).

Further examples, technical details, properties, and references about limited automata can be found in the survey by Pighizzini [Pig19].

2.2.3 Context-free grammars

In formal language theory, the most investigated classes are probably those of regular and context-free languages. The interest for them is not purely theoretical, but it is also related to their practical applications as, for instance, the definition of programming language syntax and the construction of lexical and syntactic analyzers [Aho+06].

As seen in Section 2.1, the difference between these two classes is related to the representation of recursive structures (*e.g.*, nested brackets, nested blocks in programming languages, arithmetic expressions in infix notations) which is possible in context-free languages but not in regular ones.

Context-free languages are generated by context-free grammars, that are formally defined as follows:

Remark 2.2. *A* context-free grammar *(*CFG*, for short) is a formal grammar* $\mathcal{G} = \langle V, \Sigma, P, S \rangle$*, in which* $V \cap \Sigma = \emptyset$ *and productions in* P *are of the form* $A \to \alpha$*, where* $A \in V$ *and* $\alpha \in (V \cup \Sigma)^*$*.*

It is possible to notice that, in context-free grammars, each variable represents a language. So, for each $A \in V$, we shall denote as $\mathcal{G}_{|A} = \langle V, \Sigma, P, A \rangle$ the grammar obtained from $\mathcal{G}$ by taking A as initial symbol. Hence $L\left(\mathcal{G}_{|A}\right)$ is the language generated starting from A.

In order to restrict the number of possible choices when deriving a string, it is often useful to replace the given derivation by the *leftmost derivation*: such derivation always rewrites the leftmost variable in the given sentential form. A leftmost derivation will be denoted by $\underset{lm}{\Rightarrow}$ and $\overset{\star}{\underset{lm}{\Rightarrow}}$, for exactly one and for arbitrarily many steps (including zero), respectively.

2.2.3.1 Non-self-embedding context-free grammars

The difference, in terms of grammars, between context-free and regular languages, was formalized by Chomsky, who studied the *self-embedding* property [Cho59a]. A variable A in a context-free grammar is *self-embedded* if it is able to reproduce itself in a sentential form, enclosed between two nonempty strings α and β. Formally,

Definition 2.5. *Let $\mathcal{G} = \langle V, T, P, S \rangle$ be a context-free grammar. A variable $A \in V$ is said to be* self-embedded *when there are two strings $\alpha, \beta \in (V \cup \Sigma)^+$ such that $A \overset{\star}{\Rightarrow} \alpha A \beta$. The grammar $\mathcal{G}$ is* self-embedding *(SE, for short) if it contains at least one self-embedded variable, otherwise $\mathcal{G}$ is* non-self-embedding *(NSE, for short).*

This means that the variable A can generate a "true" recursion that needs an auxiliary memory (typically a stack) to be implemented (in contrast with tail or head recursions, corresponding to the cases in which α or β are empty, respectively, that can be easily eliminated). Chomsky proved that context-free grammars *without* self-embedded variables, namely *non-self-embedding grammars*, only generate regular languages, *i.e.*, they are no more powerful than finite automata [Cho59a, Cho59b].

Hence, the "true" recursion given by self-embedded variables is the additional capability which makes the class of context-free languages larger than the class of regular ones.

It is worthwhile to mention that, when considering unary alphabets, the classes of context-free and regular languages collapse [GR62]. Hence, context-free grammars with unary terminal alphabets generate regular languages. So one could ask what happens if we consider context-free grammars where the only variables which are allowed to be self-embedded are those generating only unary strings. Let us call such grammars *quasi-non-self-embedding*.

Definition 2.6. *Let $\mathcal{G}$ be a context-free grammar. $\mathcal{G}$ is* quasi-non-self-embedding *(*qNSE*, for short) when each self-embedded variable generates a unary language.*

Note that in qNSE grammars the self-embedded variables could generate different unary languages on different symbols of T.

The results given by Chomsky on NSE grammars have been extended to qNSE grammars by Andrei, Cavadini, and Chin, who proved that also qNSE grammars (called *one-letter factorizable*), generate only regular languages [ACC03].

We point out that, as shown by Anselmo, Giammarresi, and Varricchio in 2002, given a grammar $\mathcal{G}$ it is possible to decide in polynomial time whether or not it is NSE [AGV02]. The proof is based on the analysis of the production graph T of $\mathcal{G}$. In particular, if each SCC of T contains either only *left-linear* or *right-linear* variables, *i.e.*, such that each production having a variable A of the SCC as a left-end side is either of the form $A \to wB$ ($A \to Bw$, resp.), or of the form $A \to w$, for some $B \in V$ and $w \in \Sigma^*$, then $\mathcal{G}$ is a NSE grammar.

With an easy modification, the same technique can be used to decide if $\mathcal{G}$ is qNSE.

2.2.4 Pushdown automata

As seen in the previous section, the difference between the two smallest classes of the Chomsky hierarchy resides in the fact that context-free languages can represent recursive structures.

While, on the one hand, such a difference can be emphasized in terms of grammars, on the other hand, in terms of recognizers, the natural counterpart to context-free grammars

are *pushdown automata*, that can be seen as particular Turing machines, that are allowed to scan the input tape in a one-way fashion and can use an auxiliary memory that is organized as a *pushdown* (or *stack*), *i.e.*, the memory structure which allows to implement the recursion, in which the information is stored and recovered in a "last in–first out" way. The stack can be imagined as a data structure that grows from the *bottom* to the *top*, in a vertical way. This means that, at each step of computation, the machine can access (*read*) *only* the last information that was stored on the top of the pushdown, it can delete such information (*pop*), or it can enter additional information on the top of the stack (*push*)[2].

When a push operation is performed to save some symbol γ on the stack, the previous topmost symbol becomes the second one from the top, and the device cannot access that symbol until a pop move is used to remove γ from the top of the pushdown. We shall represent the content of the stack as a string, with the convention that the leftmost symbol represents the top of the stack, while the rightmost symbol is the bottom of the pushdown.

The stack gives this device the capability to implement recursive procedures. Hence, due to its capability to recognize recursive structures in languages, this model characterizes the class of context-free languages [Cho62].

Formally, a pushdown automaton is defined as follows:

Definition 2.7. *A* pushdown automaton *(*PDA*, for short) is a tuple* $\mathcal{M} = \langle Q, \Sigma, \Gamma, \delta, q_I, Z_0, F \rangle$ *where* Q *is the finite* set of states, Σ *is the* input alphabet, Γ *is the* pushdown alphabet, δ *is the* transition function, $q_I \in Q$ *is the* initial state, $Z_0 \in \Gamma$ *is the* start symbol, $F \subseteq Q$ *is the* set of final states.

At the beginning of the computation the automaton starts with the input head being on the first cell of the input tape, the pushdown store containing only a special symbol Z_0 denoting the bottom of the stack, and the finite control storing the initial state.

[2]Notice that in the literature there exists a generalization of pushdown automata, whose name is *stack automata*, in which, besides the operations push and pop, it is allowed to inspect — in a read-only way — the contents stored below the symbol on top [HU69]. Throughout this thesis we study only pushdown automata, however we shall use both "pushdown" and "stack" to refer to their memory structure.

The input of a PDA is accepted if and only if the automaton reaches a final state $q_F \in F$, the pushdown store contains only Z_0 and all the input has been scanned.

On classical text-books, the transition function δ is usually defined as a mapping from $Q \times (\Sigma \cup \{ \varepsilon \}) \times \Gamma$ to the finite subsets of $Q \times \Gamma^*$. So, at each step, according to the current state, the symbol scanned by the input head, and the symbol on the top of the pushdown, the device (nondeterministically) selects the next state, moves the input head to the right (unless a ε-move is performed), and replaces the symbol at the top of the stack with a new string.

In order to study this model and some of its restrictions, it will be useful to make the following assumptions about PDAs.

First of all, it is possible to transform each PDA in the above-given form in an equivalent PDA that pushes exactly one symbol on the stack for each push operation. Formally, each transition $(p, Y_1 Y_2 \cdots Y_n) \in \delta(q, a, X)$ such that $n > 1$, can be replaced by the following transitions:

1. $\delta(q, a, X) = \{ (q', \varepsilon) \}$,
2. $\delta(q', \varepsilon, \gamma) = \{ (p_n, Y_n \gamma) \}$, for each $\gamma \in \Gamma$,
3. $\delta(p_{i+1}, \varepsilon, Y_{i+1}) = \{ (p_i, Y_i Y_{i+1}) \}$, for $i = n-1, \ldots, 1$,
4. $\delta(p_1, \varepsilon, Y_1) = \{ (p, Y_1) \}$.

With this transformation, the topmost symbol is popped off the stack with Transition 1, and then the symbols of the string $Y_1 Y_2 \cdots Y_n$ are pushed on the stack, one per transition (Transitions 3). We point out that the rightmost symbol of $Y_1 Y_2 \cdots Y_n$ is pushed regardless the new topmost symbol (Transition 2). Finally, Transition 4 is used to reach p without performing any operation on the stack. Notice that, if the first pushed symbol of $Y_1 Y_2 \cdots Y_n$ is X, *i.e.*, $Y_n = X$, then it is not necessary to pop X off the stack. In this case Transitions 1 and 2 can be replaced by $\delta(q, a, X) = \{ (p_n, X) \}$.

It can be observed that the number of auxiliary states p_i, $i = 1, \ldots, n$ introduced to split up the transition $(p, Y_1 Y_2 \cdots Y_n) \in \delta(q, a, X)$ is linear in the length of $Y_1 Y_2 \cdots Y_n$.

Hence, this transformation requires an increase of number of states linear in the length of the string pushed with each replaced transition.

Moreover, we can assume, without loss of generality, that if the automaton moves the input head, then no operations are performed on the stack. In such a way, the transitions manipulating the pushdown store are clearly distinguished from those reading the input tape.

This is done by replacing each transition $(p, Y) \in \delta(q, a, X)$, with $p, q \in Q$, $a \in \Sigma \setminus \{ \varepsilon \}$, and $X, Y \in \Gamma$, with two transitions

$$\delta(q, a, X) = \{ (p', X) \} \text{ and } \delta(p', \varepsilon, X) = \{ (p, Y) \}.$$

It is an easy observation that this transformation only requires a constant increase of the number of states of the original automaton.

The *transition function* δ of a PDA $\mathcal{M}$ in this form can be written as

$$\delta : Q \times (\Sigma \cup \{ \varepsilon \}) \times \Gamma \to 2^{Q \times (\{ -, \mathtt{pop} \} \cup \{ \mathtt{push}(A) \mid A \in \Gamma \})}.$$

In particular, for $q, p \in Q$, $A, B \in \Gamma$, $\sigma \in \Sigma$, $(p, -) \in \delta(q, \sigma, A)$ means that the PDA $\mathcal{M}$, in the state q, with A at the top of the stack, by consuming the input σ, can reach the state p without changing the stack contents; $(p, \mathtt{pop}) \in \delta(q, \varepsilon, A)$ ($(p, \mathtt{push}(B)) \in \delta(q, \varepsilon, A)$, $(p, -) \in \delta(q, \varepsilon, A)$, respectively) means that $\mathcal{M}$, in the state q, with A at the top of the stack, without reading any input symbol, can reach the state p by popping off the stack the symbol A on the top (by pushing the symbol B on the top of the stack, without changing the stack, respectively).

Notice that in any accepting computation the occurrence of the start symbol Z_0 at the bottom of the stack is never removed, otherwise the next move would be undefined, so halting in a non-accepting configuration.

Constant-Height Pushdown Automata. We shall consider a restriction of the model in which the capacity of the pushdown store is bounded by some constant $h \in \mathbb{N}$. Hence,

the number of their possible configurations is finite. This implies that they are no more powerful than finite automata.

Definition 2.8. *For $h \in \mathbb{N}$, a* pushdown automaton of height h *(h-*PDA*) is a pushdown automaton $\langle Q, \Sigma, \Gamma, \delta, q_I, Z_0, F \rangle$ in which, during the computation, the number of symbols contained in the pushdown store besides the start symbol Z_0 must be less than h. In particular, it is possible to perform* `push` *transitions if the height of the pushdown store is less than h.*

We point out that the height of a PDA $\mathcal{A}$ in a given configuration is the number of symbols in the pushdown store *besides* the start symbol.

As for pushdown automata, this model accepts an input word $w \in \Sigma^*$ if, starting from the initial state q_I with a pushdown store containing only the start symbol Z_0, being at height 0, it can eventually reach the accepting state q_F, after having read all the input symbols.

2.2.5 Finite state automata

Finite state automata are the simplest model we are going to consider, and since they are the standard model used to describe regular languages, in the thesis we shall study relationship of above-defined models with these devices.

These machines have been originally introduced to model brain functions (as *neural nets*) [MP43], *circuits* [Huf54, Mea55] and as "*sequential machines*" [Moo56].

This model can be seen as restricted Turing machines that are not allowed to use other memory than its finite control, *i.e.*, it cannot change the contents of any cell of the tape during the computation, nor it has an auxiliary memory (as in the case of pushdown automata). It can also be noticed that such a description corresponds to the definition of 0-limited automata.

Finite automata, in all their settings, namely deterministic, nondeterministic, one-way and two-way, characterize the class of regular languages.

We start by introducing the model in the more general setting (two-way nondeterministic finite automata) and then we consider the one-way and the deterministic versions of

this device.

Definition 2.9. *A* two-way nondeterministic finite automaton *(*2NFA*, for short) is a tuple* $\mathcal{A} = \langle Q, \Sigma, \delta, q_I, F \rangle$*, where* Q *is a finite* set of states, Σ *is a finite* input alphabet, $q_0 \in Q$ *is the* initial state, $F \subseteq Q$ *is a* set of final states, *and* $\delta : Q \times \Sigma_{\triangleright\triangleleft} \to 2^{Q \times \{-1,0,+1\}}$ *is a* nondeterministic transition function, *where* $\Sigma_{\triangleright\triangleleft}$ *denotes the set* $\Sigma \cup \{\triangleright, \triangleleft\}$*, and* $\triangleright, \triangleleft \notin \Sigma$ *are the* left *and the* right endmarkers*, respectively.*

A 2NFA $\mathcal{A}$ is said to be *deterministic* (2DFA, for short), whenever $\#\delta(q, \sigma) \leq 1$, for any $q \in Q$ and $\sigma \in \Sigma_{\triangleright\triangleleft}$. It is called *one-way* if its head can never move backward, *i.e.*, if no transition returns -1 as second component. By 1NFAs and 1DFAs (or simply NFAs and DFAs, when the one-way direction of the head is clear from the context) we denote one-way nondeterministic and deterministic finite automata, respectively.

2.3 Descriptional Complexity of Formal Systems

Descriptional complexity is an area of formal languages and automata theory whose goal is the investigation of the relationship between the sizes of the representations of equivalent formal systems.

In the previous section a few equivalent models characterizing the class of regular languages have been defined by limiting some resources of more general models. Therefore, the main question we are interested about is *"How much does the limitation of one resource cost in terms of another resource,* i.e., *what are the upper and lower bounds of such costs?"* [Gol+02].

Since we are interested in comparing the sizes of the descriptions of devices and formal systems, for each model under consideration we evaluate its *size* as the total number of symbols used to describe it, or, in other words, to write down its description. In the following, for each considered model, we give a more precise definition of the measure of complexity considered. In particular, given a computational model $\mathcal{A}$, we denote its size by $\texttt{size}(\mathcal{A})$.

We remark that through the thesis we shall consider machines over a *fixed input alphabet* Σ.

Turing machines. Given a deterministic Turing machine $\mathcal{T} = \langle Q, \Sigma, \Gamma, \delta, q_0, F \rangle$, the size of $\mathcal{T}$ is bounded by a polynomial in the number of states and of working symbols. In particular, $\texttt{size}(\mathcal{T}) \in \Theta(\#Q \cdot \#\Gamma \cdot \log(\#Q \cdot \#\Gamma))$.

We remark that such a measure of complexity holds for the bounded, weight reducing, and linear-time Turing variants as well, while, if $\mathcal{T}$ is nondeterministic, $\texttt{size}(\mathcal{T}) \in \Theta((\#Q)^2(\#\Gamma)^2)$.

Limited automata. For d-limited automata, the size depends on the number of states and on the cardinality of the working alphabet.

Hence, the size of a 1-LA $\mathcal{A} = \langle Q, \Sigma, \Gamma, \delta, q_I, F \rangle$, is $\Theta((\#Q)^2(\#\Gamma)^2)$.

We remind the reader that, if not otherwise specified, limited automata are considered to be nondeterministic.

In case $\mathcal{A}$ is deterministic, $\texttt{size}(\mathcal{A}) \in \Theta(\#Q \cdot \#\Gamma \cdot \log(\#Q \cdot \#\Gamma))$.

Pushdown automata. The size of a PDA $\mathcal{M} = \langle Q, \Sigma, \Gamma, \delta, q_I, Z_0, F \rangle$ in the above-defined form, is $\Theta((\#Q)^2(\#\Gamma)^2)$, namely a polynomial in the cardinalities of the set of states and of the pushdown alphabet.

Notice that if we consider PDAs in different forms, as that given in [HU79] in which any push operation can replace the top of the pushdown by a string of symbols, to define the size we have to take into account also the number of symbols that can be pushed on the store in one single operation. However, PDAs in that form can be turned in the form we consider through this thesis with a polynomial increase in size (and by preserving the property of being constant height). For a further discussion on this point we address the reader to [Bed+14].

h-PDAs, instead, can be replaced by equivalent standard PDAs without the built-in limit on pushdown size, by counting in the finite control the pushdown height,

with an increase in the number of states that is linear in h. For this reason, the size of an h-PDA over a fixed input alphabet Σ is given by a polynomial in $\#Q$, $\#\Gamma$, and h [GMP10].

Context-free grammars. For a context-free grammar $\mathcal{G} = \langle V, \Sigma, P, S \rangle$, as size of $\mathcal{G}$ we consider the total number of symbols used to specify it. *i.e.*, the number of symbols used to write down its productions. Therefore, for fixed Σ and V, $\texttt{size}(\mathcal{G})$ is defined as $\sum_{(A \to \alpha) \in P} (2 + |\alpha|)$, *cf.* [Kel84]. Since in general we cannot assume that V is fixed, the previous value is multiplied by $\log \#V$.

Finite automata. To measure the size of a finite automaton, since no writings are allowed (neither on the tape nor on an auxiliary memory) and hence no working alphabet is provided, we consider the cardinality of the state set.

More precisely, in the case of (one-way and two-way) nondeterministic finite automata, the size is linear in the number of instructions and states, which is bounded by a function quadratic in the number of states. So, the size of a NFA $\langle Q, \Sigma, \delta, q_I, F \rangle$ is $\Theta((\#Q)^2)$. Instead, in the deterministic case (for both the one-way and two-way models), the size of a DFA $\mathcal{A} = \langle Q, \Sigma, \delta, q_I, F \rangle$ is bounded by a function linear in the number of states. Therefore, $\texttt{size}(\mathcal{A}) \in \Theta(\#Q \cdot \log \#Q)$.

The main goal of this thesis is to study the relationships between the size of these different models.

In particular, given two different classes of computational models $\mathcal{M}_1$ and $\mathcal{M}_2$ characterizing the regular languages, there are natural questions we are interested in:

- Does a function F exist, such that for all the regular languages $L \in \text{REG}$, the size of the smallest device of type $\mathcal{M}_1$ for L is upper bounded by the function F of the size of the smallest equivalent device of type $\mathcal{M}_2$, *i.e.*,

$$\min\{\, \texttt{size}(\mathcal{A}) \mid \mathcal{A} \text{ is a device in } \mathcal{M}_1 \text{ accepting } L \,\} \leq F(\min\{\, \texttt{size}(\mathcal{A}) \mid \mathcal{A} \text{ is a device in } \mathcal{M}_2 \text{ accepting } L \,\})?$$

If F exists, it is an *upper bound* for the increase (*blow-up*) in complexity when changing from a minimal model of type $\mathcal{M}_2$ for an arbitrary regular language to an equivalent minimal model of type $\mathcal{M}_1$.

- Do an infinite family of distinct languages $(L_i)_{i=0}^{\infty}$, $L_i \in \text{REG}$, $i \in \mathbb{N}$, and a function f exist, such that for all $i \in \mathbb{N}$, the size of the smallest device of type $\mathcal{M}_1$ for L is lower bounded by the function f of the size of the smallest equivalent device of type $\mathcal{M}_2$, *i.e.*,

$$\min\{\, \texttt{size}(\mathcal{A}) \mid \mathcal{A} \text{ is a device in } \mathcal{M}_1 \text{ for } L_i \,\} \geq f(\min\{\, \texttt{size}(\mathcal{A}) \mid \mathcal{A} \text{ is a device in } \mathcal{M}_2 \text{ for } L_i \,\})?$$

  If f exists, it is a *lower bound* for the increase in complexity when changing from a minimal model of type $\mathcal{M}_2$ to an equivalent minimal model of type $\mathcal{M}_1$ for infinitely many languages.

If there is no recursive function upper bounding the trade-off between two computational models $\mathcal{M}_1$ and $\mathcal{M}_2$, we say that the trade-off is *non-recursive*. Furthermore, if the lower and the upper bounds coincide, we say that the bound is *tight* or *optimal*.

For more details about the area of descriptional complexity see, *e.g.*, the surveys by Goldstine et al., Holzer and Kutrib [Gol+02, HK10].

A classical example in the area of descriptional complexity is the relationship between the size of deterministic and nondeterministic finite automata.

It is well known that, for finite automata, the nondeterminism does not add power. However, the elimination of nondeterminism, obtained by applying the *subset* (or *powerset*) *construction*, cost exponential [RS59]. The subset construction gives an exponential upper bound for the elimination of nondeterminism from one-way finite automata, and it is possible to prove that such a bound is tight. In fact, we shall show an example in which, given an infinite family of languages $(L_i)_{i=0}^{\infty}$, the size of each deterministic finite

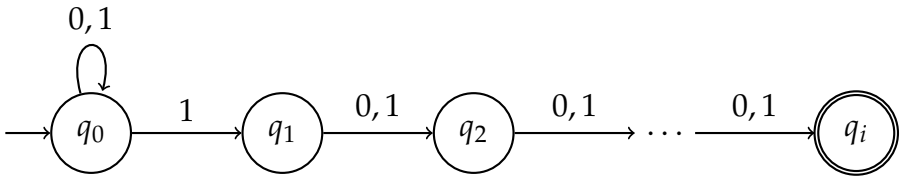

Figure 2.3: A nondeterministic finite automaton accepting the language composed by strings in which the i-th symbol from the end is a 1.

automaton accepting L_i, for $i \in \mathbb{N}$, is at least exponential in the size of the equivalent nondeterministic finite automaton.[3]

Example 2.5 ([HMU01])**.** Let us consider the following language:

$$L_i = \left\{ w \in \{0,1\}^* \mid \text{the } i\text{-th to last symbol of } w \text{ is } 1 \right\}.$$

A 1NFA $\mathcal{A}_i$ accepting L_i (represented in the transition graph in Figure 2.3) could work as follows. It moves the input head forward on the input tape, until reaching the i-th symbol from the end, that is detected by performing a nondeterministic guess. If the i-th last symbol is a 1, the automaton performs a transition from q_0 to q_1, and checks if the length of the remaining part of the input string is $i-1$ (this is done by using states from q_1 to q_i). Therefore, the number of states for a NFA accepting L_i is $i+1$.

On the other hand, a DFA accepting L_i cannot guess which is the i-th symbol from the end of the input string. So, intuitively, it has to "*remember*" the last window of i symbols, saving it by using its finite control, and, when the end of the input is reached, checking that the first symbol of the current window is 1. It is easy to see that the number of the possible windows of length i is 2^i, and each of them is stored by using a state of the 1DFA, thus implying a number of states equal to 2^i. ■

[3]For a fixed i, the language presented has an $i+1$-state NFA accepting it, while the equivalent DFA needs at least 2^i states. The choice of such a language is done for ease of presentation. However, Meyer and Fischer showed that, for each n, there exists a language that can be accepted by an n-state NFA but for which each DFA requires at least 2^n states [MF71].

It is also known that the capability of moving the input head back and forth along the tape does not change the recognizing power of this model. In fact, Shepherdson and Rabin and Scott, independently, in 1959, prove this result by giving a two constructions that are based on the analysis of the *crossing sequences*, *i.e.*, the moves of the input head of the automata through the tape cells [She59, RS59]. Both the constructions, given a two-way finite automaton, return an equivalent one-way deterministic finite automaton whose size is exponential in the square of the size of the automaton given in input.

Nevertheless, the question about the cost of the elimination of the nondeterminism from 2FA is still open. In particular, Sakoda and Sipser in 1978 raised a question about the cost of the optimal simulation of 1NFAs and 2NFAs by 2DFAs. Sakoda and Sipser conjectured that such a cost is exponential.

The question has been solved in some special cases that can be grouped in three classes: by considering restrictions on the simulating machines (*e.g.*, *sweeping* [Sip80], *oblivious* [HS03], *few reversals* two-way deterministic finite automata [Kap13]), by considering restrictions on languages (*e.g.*, *unary case* [GMP03]), by considering restrictions on the simulated machines (*e.g.*, *outer-nondeterministic* automata [GGP14, KP15]). However, in spite of all attempts, in the general case the question remains open.

The importance of this open problem is supported by its similarity to the well-known P $\stackrel{?}{=}$ NP problem [SS78] and by relationships with LOGSPACE $\stackrel{?}{=}$ NLOGSPACE question [BL77, GP11, Kap14b, KP15]. (See [Pig13] for further details and references.)

3

Non-Self-Embedding Grammars

As mentioned in Section 2.2.3.1, Chomsky in 1959 investigated the self-embedding property of context-free grammars, and proved that non-self-embedding grammars only generate regular languages.

The proof of this result given by Chomsky is constructive: it provides a method for obtaining a finite automaton equivalent to a given non-self-embedding grammar [Cho59a, Cho59b]. A different constructive proof of the same result was given by Anselmo, Giammarresi, and Varricchio, who showed that it is possible to decompose non-self-embedding grammars into regular grammars and then to iteratively apply regular substitutions in order to obtain equivalent finite automata [AGV02]. In the same paper, the authors also proved that the size gap between non-self-embedding grammars and equivalent finite automata is at least exponential, by showing the existence of a language (defined over a one-letter alphabet) described by a non-self-embedding grammar of size $O(s)$ for which any equivalent nondeterministic finite automaton requires 2^s many states.

In this chapter we continue the investigation of the relationships between the sizes of non-self-embedding grammars, together with the quasi-non-self-embedding extension, and of equivalent finite automata.

It is worthwhile to mention that, in 1971, Meyer and Fischer proved that for any recursive function f and arbitrarily large integer n, there exists a context-free grammar whose description has size n and which generates a regular language, such that any equivalent

finite automaton requires at least $f(n)$ states [MF71]. This means that it is not possible to obtain a recursive bound relating the size of context-free grammars generating regular languages with the number of states of equivalent deterministic finite automata. It is important to notice that the result of Meyer and Fischer was obtained by considering grammars with a two-letter terminal alphabet. The unary case was studied in 2002 by Pighizzini, Shallit, and Wang, who obtained optimal recursive bounds [PSW02].

In this chapter we show that, also in the case of non-self-embedding grammars, the bounds are recursive, independently on the alphabet size. In particular, by inspecting and refining the construction presented by Anselmo, Giammarresi, and Varricchio, we show that each non-self-embedding grammar of size s can be converted into equivalent nondeterministic and deterministic automata with $2^{O(s)}$ and $2^{2^{O(s)}}$ states, respectively. We also present a family of languages that witness that these gaps cannot be reduced.

Moreover, we prove that the size costs of the conversion of quasi-non-self-embedding grammars into equivalent nondeterministic and deterministic finite automata are the same of the conversion of non-self-embedding grammars.

The results shown in this chapter have been presented in [PP17].

3.1 Preliminaries

In the paper by Anselmo, Giammarresi, and Varricchio, the authors showed the existence of a family of unary languages that witnesses the exponential size gap between non-self-embedding grammars and finite automata. In order to show an example of NSE grammar, we now present such a family of languages and the grammar used to describe it.

Example 3.1 ([AGV02])**.** Let us consider the family of languages $(U_n)_{n=0}^{\infty}$, such that, for each n, $U_n = \{\, a^{2^n} \,\}$, that is the singleton composed by the unary string of length 2^n.

Fixed an integer n, U_n is generated by the NSE grammar

$$\mathcal{G}_n = \langle \{\, A_0, \ldots, A_n \,\}, \{\, a \,\}, P, A_n \rangle$$

where the set P of productions contains

- $A_0 \to a$ and
- $A_i \to A_{i-1}A_{i-1}$, for each $i = 1, \ldots, n$.

It is possible to see that the variable A_i generates the string a^{2^i}, for $i = 0, \ldots, n$. Hence, the language generated by $\mathcal{G}_n$ is U_n.

Moreover, the size of $\mathcal{G}_n$ is linear in the parameter n, while the minimum DFA $\mathcal{A}_n$ accepting U_n has $2^n + 1$ states. In fact, if $\mathcal{A}_n$ had less than $2^n + 1$ states, for the *pigeons' hole principle*, a computation path on the string a^{2^n} would pass more than once in at least one state, thus implying the existence of a cycle in the transition graph of $\mathcal{A}_n$, and so that $\mathcal{A}_n$ would recognize an infinite language. However, the same lower bound can be proved for 2NFAs. Indeed, by adapting the $n \to n + n!$ *method*, originally derived for Turing machines operating by using an auxiliary working tape of size bounded by a function sublogarithmic in the input length [HR89, Gef91], one can prove that if a 2NFA accepts a^k with less than k states, it must also accept $a^{k+k!}$. ■

The family of languages $(U_n)_{n=0}^{\infty}$ will be used often throughout this dissertation to witness the bounds on the size of models.

We now introduce some technical results and definitions that will be used in the chapter.

Given two alphabets Σ and Γ, a *substitution* is a function φ mapping each letter $\gamma \in \Gamma$ into a language $\varphi(\gamma) \subseteq \Sigma^*$. The substitution φ can be extended to strings and languages in a standard way. The substitution φ is said to be *regular* if $\varphi(\gamma)$ is a regular language for each $\gamma \in \Gamma$. It is well known that regular substitutions preserve regularity.

Lemma 3.1. *Let Σ and Γ be two alphabets and $\varphi : \Gamma \to 2^{\Sigma^*}$ be a* regular substitution, i.e., *a function mapping each letter $\gamma \in \Gamma$ into a regular language $\varphi(\gamma) \subseteq \Sigma^*$. If $L \subseteq \Gamma^*$ is a regular language accepted by an* NFA *$\mathcal{M}$ with n_1 states, and each language $\varphi(\gamma)$ is accepted by an* NFA *$\mathcal{M}_\gamma$ with at most n_2 states, then the language $\varphi(L)$ is accepted by an* NFA *$\mathcal{N}$ with $O(n_1 n_2)$ states. Furthermore, for each nontrivial* SCC *in the transition graph of $\mathcal{M}$ there exists a corresponding "expanded"* SCC *in $\mathcal{N}$,* (i.e., *if starting from the initial state of $\mathcal{M}$ and reading a*

string $x \in \Gamma^*$ a state in a nontrivial SCC *is reached, then starting from the initial state of $\mathcal{N}$ and reading $\varphi(x)$ a state in the corresponding* SCC *is reached) while each remaining nontrivial* SCC *in $\mathcal{N}$ is an isomorphic copy of some nontrivial* SCC *occurring in automata $\mathcal{M}_\gamma$, $\gamma \in \Gamma$.*

Proof. The automaton $\mathcal{N}$ can be obtained by "substituting" each automaton $\mathcal{M}_\gamma$ in $\mathcal{M}$, namely by replacing in $\mathcal{M}$ each transition from a state p to a state q on the symbol γ with a copy of the NFA $\mathcal{M}_\gamma$. More precisely, using ε-moves, in $\mathcal{N}$ the initial state of the copy of $\mathcal{M}_\gamma$ can be reached from the state p while the state q can be reached from each final state in the copy. If $\mathcal{M}$ has n_1 states (hence $O(n_1^2)$ transitions) and each $\mathcal{M}_\gamma$ has at most n_2 states, then the resulting NFA $\mathcal{N}$ has $O(n_1^2 n_2)$ states. This number can be reduced to $O(n_1 n_2)$ by using, for each state p, only one copy of $\mathcal{M}_\gamma$ for all outgoing transitions from p on the symbol γ, and connecting with ε-transitions the final states of the copy to all the states q which in $\mathcal{M}$ are reachable from p by transitions on γ. Finally, ε-transitions can be removed in a standard way, without increasing the number of states.

We notice that in this construction, from each nontrivial SCC of $\mathcal{M}$ we obtain a nontrivial SCC in $\mathcal{N}$. Each other nontrivial SCC in $\mathcal{N}$ is created as a copy of a nontrivial SCC in an automaton $\mathcal{M}_\gamma$ when a transition of $\mathcal{M}$ on a symbol $\gamma \in \Gamma$ connecting two states of $\mathcal{N}$ not belonging to a same SCC is replaced by the automaton $\mathcal{M}_\gamma$. □

3.1.1 Standard constructions: converting grammars to automata

The *production graph* $\mathcal{P}(\mathcal{G})$ of a context-free grammar $\mathcal{G} = \langle V, \Sigma, P, S \rangle$ is a directed graph which has V as vertex set and contains an edge from A to B, $A, B \in V$, if and only if there is a production $A \to \alpha B \beta$ in P, for $A, B \in V$ and some $\alpha, \beta \in (V \cup \Sigma)^*$. The strongly connected components of the production graph induce a partial order on variables: a variable A is *smaller than* B if there exists a path from A to B and no path from B to A.

The grammar $\mathcal{G}$ is said to be *right-linear* (*left-linear*, resp.), if each production in P is either of the form $A \to wB$ ($A \to Bw$, resp.), or of the form $A \to w$, for some $A, B \in V$, $w \in \Sigma^*$. It is well known that right- or left-linear grammars generate exactly the class of regular languages.

Lemma 3.2. *Each left-linear grammar of size s can be converted into an equivalent right-linear grammar of size $O(s)$.*

Proof. Let $\mathcal{G} = \langle V, \Sigma, P, S \rangle$ be a left-linear grammar. It is possible to obtain an equivalent right-linear grammar $\mathcal{G}' = \langle V', \Sigma, P', S' \rangle$ whose size is linear in the size of $\mathcal{G}$, in which the set of variables $V' = V \cup \{ S' \}$ has one more variable $S' \notin V$, and the set P' contains the following productions:

- $B \to w$, for each $S \to Bw$ in P,
- $B \to wA$, for each $A \to Bw$ in P (including $S \to Bw$),
- $S' \to wA$, for each $A \to w$ in P.[1]

Notice that if we apply the above transformation to the grammar $\mathcal{G}_{|\hat{S}}$ obtained by changing the initial symbol of $\mathcal{G}$ into $\hat{S} \in V$, then the set of productions of the resulting grammar can be different from P'. □

From each left-linear (or right-linear) grammar $\mathcal{G}$ of size s it is possible to obtain an equivalent NFA with $O(s)$ many states. More precisely, we can prove the following result:

Lemma 3.3. *Let $\mathcal{G} = \langle V, \Sigma, P, S \rangle$ be a right-linear (left-linear, resp.) grammar of size s. Then, the following holds:*

1. *There exists an equivalent* NFA *$\mathcal{M}$ with $O(s)$ states.*
2. *If the production graph of $\mathcal{G}$ is strongly connected and its unique* SCC *is not trivial, then the transition graph of $\mathcal{M}$ contains one nontrivial* SCC*, and this* SCC *includes the initial (final, resp.) state of $\mathcal{M}$.*

Proof. First of all, if $\mathcal{G}$ is a left-linear grammar, then, by Lemma 3.2, it is possible to convert it into a right-linear grammar whose size is linear in s. So, let us suppose $\mathcal{G}$ right-linear and let us recall how to convert $\mathcal{G}$ to an equivalent NFA $\mathcal{M}$. By standard construction, $\mathcal{M}$ is obtained by:

[1] For our purposes, it is useful to observe that we can rename the variables in the resulting grammar $\mathcal{G}'$, in such a way that the name of the start symbol is S, as in the original grammar.

i. Creating a state q_A for each variable $A \in V$ and a final state q_F.

ii. Adding a path labeled by w (with relative intermediate states) from q_A to q_B for each production of the form $A \to wB$, $w \in \Sigma^*$.

iii. Adding a path labeled by w (with relative intermediate states) from q_A to q_F for each production of the form $A \to w$, $w \in \Sigma^*$.

iv. Setting q_S — the state representing the variable S — as initial.

The obtained graph corresponds to the transition graph of an NFA equivalent to $\mathcal{G}$.

To prove 1 it is sufficient to see that for each production $A \to \alpha$ in P, $O(|\alpha|)$ states are created. So, it is possible to conclude that the number of states of $\mathcal{M}$ is in $O(s)$.

The proof of 2 follows from the observation that, if the production graph is a unique nontrivial SCC then there exists a path from A to B and from B to A, for all $A, B \in V$. Hence, there exists a unique nontrivial SCC in $\mathcal{M}$ containing all the states q_A, $A \in V$, and the extra states introduced at step ii.; the remaining states on the paths from q_A, $A \in V$, to q_F (and the state q_F itself) form trivial components.

□

From the construction in the proof of Lemma 3.3, we can notice that, by changing the initial symbol in $\mathcal{G}$, the obtained NFA can be different (in the case of a right-linear grammar, we need to change the initial state, while in the case of left-linear grammars we have to change the final state).

Anselmo, Giammarresi, and Varricchio showed that NSE grammars admit a particular form based on a decomposition into finitely many simpler grammars, that will be now recalled.

Definition 3.1 ([AGV02])**.** *Let* $\mathcal{G}_1 = \langle V_1, \Sigma_1, P_1, S_1 \rangle$ *and* $\mathcal{G}_2 = \langle V_2, \Sigma_2, P_2, S_2 \rangle$ *be two* CFGs*, with* $V_1 \cap V_2 = \emptyset$*. The* $\oplus$-composition *of* $\mathcal{G}_1$ *and* $\mathcal{G}_2$ *is the grammar*

$$\mathcal{G} = \mathcal{G}_1 \oplus \mathcal{G}_2 = \langle V, \Sigma, P, S \rangle,$$

where $V = V_1 \cup V_2$, $\Sigma = (\Sigma_1 \setminus V_2) \cup \Sigma_2$, $P = P_1 \cup P_2$, *and* $S = S_1$.

Example 3.2. Let us consider two CFGs

$$\mathcal{G}_1 = (\{ S_1, C, D \}, \{ b, A, B \}, P_1, S_1) \text{ and } \mathcal{G}_2 = (\{ S_2, A, B \}, \{ a, b \}, P_2, S_2),$$

where P_1 contains the productions

$$S_1 \to bS_1 \mid C \qquad C \to DC \mid D \qquad D \to AB$$

and P_2 contains

$$S_2 \to A \mid B \mid \varepsilon \qquad A \to Aa \mid a \qquad B \to Bb \mid b.$$

It can be verified that $L(\mathcal{G}_1) = b^*(AB)^+$, in $\mathcal{G}_2$ the variable A generates the language a^+, the variable B generates the language b^+, and $L(\mathcal{G}_2) = a^* + b^*$.

The $\oplus$-composition of $\mathcal{G}_1$ and $\mathcal{G}_2$ is $\mathcal{G}_1 \oplus \mathcal{G}_2 = (\{ S_1, S_2, A, B, C, D \}, \{ a, b \}, P_1 \cup P_2, S_1)$, that generates the language $L(\mathcal{G}_1 \oplus \mathcal{G}_2) = b^*(a^+b^+)^+ = b^*a(a+b)^*b$, *i.e.*, the set of all the strings on $\{ a, b \}$ ending with b and in which at least one a occurs. ■

Intuitively, the grammar $\mathcal{G} = \mathcal{G}_1 \oplus \mathcal{G}_2$ generates all the strings which can be obtained by replacing in any string $w \in L(\mathcal{G}_1)$ each symbol $A \in \Sigma_1 \cap V_2$ with a string in Σ_2^* which can be derived from A in the grammar $\mathcal{G}_2$, *i.e.*, which is in the language $L(\mathcal{G}_{2|A})$. Notice that the definition of $\mathcal{G}_1 \oplus \mathcal{G}_2$ does not depend on the initial symbol S_2 of G_2, *i.e.*, by changing the initial symbol of $\mathcal{G}_2$ the resulting grammar $\mathcal{G}$ does not change. In fact, it can be observed that in Example 3.2 the variable S_2 of $\mathcal{G}_1 \oplus \mathcal{G}_2$ is never generated starting from the initial nonterminal symbol S_1.

The $\oplus$-composition is associative and preserves the non-self-embedding property of grammars. Moreover, if $\Sigma_1 \cap V_2 = \emptyset$ then $L(\mathcal{G}_1 \oplus \mathcal{G}_2) = L(\mathcal{G}_1)$. It is also possible to observe that given two CFGs $\mathcal{G}_1$ and $\mathcal{G}_2$, if $L(\mathcal{G}_1)$ is regular and, for each $A \in \Sigma_1 \cap V_2$, $L(\mathcal{G}_{2|A})$ is regular, then $L(\mathcal{G})$ is regular as well (by regular substitution). In particular, given an NFA $\mathcal{M}_1$ accepting $L(\mathcal{G}_1)$ and NFAs $\mathcal{M}_A$ accepting $L(\mathcal{G}_{2|A})$, for $A \in \Sigma_1 \cap V_2$, we can obtain an NFA $\mathcal{M}$ accepting $L(\mathcal{G}_1 \oplus \mathcal{G}_2)$ by "substituting" each automaton $\mathcal{M}_A$ in $\mathcal{M}_1$, as in the proof of Lemma 3.1.

3.2 Converting Non-Self-Embedding Grammars into Automata

Anselmo, Giammarresi, and Varricchio gave an interesting alternative proof of the above mentioned result of Chomsky on the regularity of languages generated by NSE grammars [AGV02]. In particular, they obtained the result as a consequence of the following theorem.

Theorem 3.1 ([AGV02, Thm. 2]). *For each* NSE *grammar* $\mathcal{G}$ *there exist* $g > 0$ *grammars*

$$\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_g$$

such that $\mathcal{G} = \mathcal{G}_1 \oplus \mathcal{G}_2 \oplus \cdots \oplus \mathcal{G}_g$*, where, for* $i = 1, \ldots, g$*,* $\mathcal{G}_i$ *is either left-linear or right-linear.*

The proof of Theorem 3.1 is constructive: it presents a method for obtaining grammars $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_g$ from the given NSE grammar $\mathcal{G}$. Since this method is important to obtain the state upper bounds for NFAs and DFAs equivalent to NSE grammars presented in this section and other results in the chapter, we now summarize how grammars $\mathcal{G}_1, \ldots, \mathcal{G}_g$ are obtained (for a detailed presentation of a related decomposition see [Har78, Sect. 3.5]).

- Let n be the number of SCCs in the production graph $\mathcal{P}(\mathcal{G})$.
- If $\mathcal{P}(\mathcal{G})$ is strongly connected then $\mathcal{G}$ is either a left-linear or a right-linear grammar, hence $\mathcal{G} = \mathcal{G}_1$ is regular.
- Otherwise, the SCCs of $\mathcal{P}(\mathcal{G})$ are considered in some topological order and, for each $i = 1, \ldots, g$, the grammar $\mathcal{G}_i = \langle V_i, \Sigma_i, P_i, S_i \rangle$ is defined as follows:
  - V_i is the set of variables in the i-th SCC.
  - $\Sigma_i = \Sigma \cup \bigcup_{j>i} V_j$, for technical reasons we also set $\Sigma_0 = \{ S \}$.
  - P_i is obtained by restricting P to productions whose left-hand side variables are in V_i.

– S_i is an element in $V_i \cap \Sigma_{i-1}$.

Since $\mathcal{P}(\mathcal{G}_i)$ is strongly connected, $\mathcal{G}_i$ is either a left-linear or a right-linear grammar, for $i = 1, \ldots, g$.

Example 3.3. Let us consider the NSE grammar $\mathcal{G} = (\{ S, A, B, C, D \}, \{ a, b \}, P, S)$, whose productions are

$$S \to aA \,|\, bB \,|\, aC \qquad A \to aS \,|\, aD \qquad B \to Bb \,|\, b \qquad C \to aC \,|\, bD \qquad D \to bC \,|\, bD \,|\, bB.$$

Hence, the production graph $\mathcal{P}(G)$ is composed by three SCCs: $\mathcal{C}_1$ whose nodes are S and A, $\mathcal{C}_2$ containing C and D, and $\mathcal{C}_3$ containing B. Following the construction recalled above it is possible to decompose $\mathcal{G}$ into three grammars $\mathcal{G}_1 = (\{ S, A \}, \{ a, b, B, C \}, P_1, S)$, $\mathcal{G}_2 = (\{ C, D \}, \{ a, b, B \}, P_2, C)$, and $\mathcal{G}_3 = (\{ B \}, \{ b \}, P_3, B)$, where P_1, P_2, and P_3 are composed by the productions of P having, as left-hand side, S or A, C or D, and B, respectively. ■

We are now going to estimate the size of NFAs and DFAs equivalent to the NSE grammar $\mathcal{G}$ given in Theorem 3.1. Let us suppose that the size of $\mathcal{G}$ is s. Then $g \leq s$. Furthermore, $s = s_1 + s_2 + \cdots + s_g$, where s_i is the size of $\mathcal{G}_i$, $i = 1, \ldots, g$.

Now, from each grammar $\mathcal{G}_i$, we obtain a family of NFAs $\{ \mathcal{M}_{i,A} \mid A \in \Sigma_{i-1} \cap V_i \}$, such that each NFA $\mathcal{M}_{i,A}$ has size $O(s_i)$ and recognizes the language $L\Big(\mathcal{G}_{i|A}\Big)$.

To obtain an NFA $\mathcal{M}$ accepting $L(\mathcal{G})$, we iteratively construct automata $\mathcal{M}_i$, for $i = 1, \ldots, g$, accepting $L(\mathcal{G}_1 \oplus \mathcal{G}_2 \oplus \cdots \oplus \mathcal{G}_i)$ as follows. Let us start by taking $\mathcal{M}_1 = \mathcal{M}_{1,S}$. For $i > 1$, the automaton $\mathcal{M}_i$ is obtained by substituting in the automaton $\mathcal{M}_{i-1}$ each transition labeled by $A \in \Sigma_{i-1} \cap V_i$ with the NFA $\mathcal{M}_{i,A}$, as explained in the proof of Lemma 3.1.

At the end of this process, we finally obtain $\mathcal{M} = \mathcal{M}_n$, which accepts $L(\mathcal{G})$ and has $O(s_1 s_2 \cdots s_g)$ many states. Since $s_1 + \cdots + s_g = s$, we get that $s_1 \cdots s_g = 2^{\log(s_1 \cdots s_g)} = 2^{\log s_1 + \cdots + \log s_g} \leq 2^s$. Hence, we conclude that $\mathcal{M}$ has $2^{O(s)}$ many states. Considering also the cost of the conversion of NFAs into equivalent DFAs, we obtain the following:

Theorem 3.2. *Let $\mathcal{G}$ be an* NSE *grammar of size s. Then there exist an* NFA *and a* DFA *accepting $L(\mathcal{G})$ with $2^{O(s)}$ and $2^{2^{O(s)}}$ states, respectively.*

3.3 Optimality

In this section we prove that the exponential and double exponential state upper bounds for the conversion of NSE grammars into NFAs and DFAs given in Theorem 3.2 cannot be reduced. To this aim, we now introduce a family $(L_h)_{h=1}^{\infty}$ of witness languages.

Given an integer $h > 0$, consider the language $L_h \subseteq \{ a, b \}^*$ defined as the set of strings composed of k blocks $w_1 w_2 \cdots w_k$ each of length h, for some $k > 1$, such that the last block w_k is the reverse of one of the first $k-1$ blocks. Formally,

$$L_h = \Big\{ w_1 w_2 \cdots w_{k-1} w_k \,|\, k > 1, w_i \in \{ a, b \}^h, i = 1, \ldots, k, \text{ and } \exists j, 1 \leq j < k, \text{ s.t. } w_j = w_k^R \Big\}.$$

Let us define the grammar $\mathcal{G}_h = \langle V, \Sigma, P, S \rangle$ generating L_h, with terminals $\Sigma = \{ a, b \}$, variables $V = \{ S, C_1, \ldots, C_h, A_1, \ldots, A_h \}$, and the following productions in P:

- $S \to C_1 A_1 \mid A_1$
- $C_i \to a C_{i+1} \mid b C_{i+1}$ for $1 \leq i < h$
- $C_h \to a \mid b \mid a C_1 \mid b C_1$
- $A_i \to a A_{i+1} a \mid b A_{i+1} b$ for $1 \leq i < h$
- $A_h \to aa \mid bb \mid a C_1 a \mid b C_1 b$

It is easy to verify that only from variables C_i, for $1 \leq i \leq h$, it is possible to derive themselves in some sentential form, *i.e.*, they are in the unique nontrivial SCC of the production graph of $\mathcal{G}$, but all of them are right-linear. Hence, $\mathcal{G}$ is a NSE grammar.

Moreover, it can be observed that from the variable C_1 it is possible to derive one or more blocks of h terminal symbols, *i.e.*, for $x \in \Sigma^*$, $C_1 \stackrel{\star}{\Rightarrow} x$ if and only if $x \in (\Sigma^h)^+$. From the variable A_1, after expanding variables A_i, $i = 1, \ldots, h$, we generate sentential forms as $A_1 \stackrel{\star}{\Rightarrow} w w^R$ or $A_1 \stackrel{\star}{\Rightarrow} w C_1 w^R$ where $w \in \Sigma^h$. Then A_1 generates all terminal strings of the form $w(\Sigma^h)^* w^R$, where $w \in \Sigma^h$. Using the initial symbol S, it is possible to combine the two variables C_1 and A_1 to obtain an arbitrarily long sequence of blocks of length h

followed by the blocks w_j and w_k that enclose another arbitrarily long sequence of blocks of h symbols.

Observe that the size of the grammar $\mathcal{G}_h$ is linear in the parameter h, *i.e.* $\texttt{size}(\mathcal{G}_h) = O(h)$. Using a distinguishability argument, we are going to show that any DFA accepting L_h requires a number of states which is double exponential in h.

Let $w_1, w_2, \ldots, w_{2^h}$ be the list of all the strings in Σ^h sorted in lexicographical order, and $S = 2^{\Sigma^h}$ be the family of all the subsets of Σ^h. For each $s \in S$, we consider the string $v_s = w_{i_1} w_{i_2} \cdots w_{i_k}$, where $1 \leq i_1 < i_2 < \ldots < i_k \leq 2^h$, $k \geq 0$, and $s = \{ w_{i_1}, w_{i_2}, \ldots, w_{i_k} \}$. Given two different subsets $s', s'' \in S$, let $x \in \Sigma^h$ be a string such that $x \in (s' \cup s'') \setminus (s' \cap s'')$. Then, the string x^R distinguishes $v_{s'}$ and $v_{s''}$, *i.e.*, exactly one between $v_{s'}x$ and $v_{s''}x$ belongs to L_h. Hence, each DFA accepting L_h needs at least $\#S$ many states. This gives a 2^{2^h} lower bound for the size of any DFA accepting L_h and a 2^h lower bound for the size of any NFA accepting L_h. This allows us to conclude that the gaps given in Theorem 3.2 cannot be reduced.

For the sake of completeness we describe how an NFA $\mathcal{N}_h$ accepting L_h works.

- $\mathcal{N}_h$ starts to scan the input string by skipping the first $j-1$ blocks and by nondeterministically guessing which is the block w_j. To this aim, $\mathcal{N}_h$ uses a counter modulo h, that requires h states.

- After that, the automaton has to read and finally save the content of w_j into its finite state control. The transition graph of this part corresponds to a complete binary tree $\mathcal{B}_h$ of height h starting from the initial state and in which each of the 2^h leaves represents a string $w \in \Sigma^h$. As a consequence, the number of states required for this part is $\sum_{i=1}^{h} 2^i = 2^{h+1} - 2$.

- After reaching a leaf it is necessary to skip $k - j + 1$ blocks. Again, this can be done in a nondeterministic way by using a counter modulo h for each leaf, each of them implemented by using $O(h)$ states.

- Finally, $\mathcal{N}_h$ has to verify that the last block corresponds to the stored word. This can

be done by comparing the next input symbol with the last symbol in the string w stored in the control; if they are different, then the computation stops, otherwise the last symbol of w is removed and the computation continues in the same way. The part of the transition graph of $\mathcal{N}_h$ for this comparison corresponds to a "reversed" binary tree of height h, in which the first level is composed by the 2^h leaves of $\mathcal{B}_h$, and the last one is composed by the unique final state of $\mathcal{N}_h$ only.

From the above description, summing up the states of the automaton, it is possible to derive an upper bound on the size of the NFA $\mathcal{N}_h$ that is exponential in the size of the NSE grammar $\mathcal{G}_h$. More precisely, we can show that $\mathcal{N}_h$ can be implemented by using $h + \sum_{i=1}^{h} 2^i + 2^h(h-1) + \sum_{i=0}^{h-1} 2^i = (2+h)2^h + h - 3$ states.

In conclusion, we proved the following:

Theorem 3.3. *The size blowup from* NSE *grammars to* NFA*s (*DFA*s, resp.) is exponential (double exponential, resp.). This bound is tight.*

3.4 Converting Quasi-Non-Self-Embedding Grammars into Automata

It is well known that unary context-free languages, *i.e.*, languages generated by CFGs with a one-letter terminal alphabet, are regular [GR62]. Hence, this holds even for unary grammars containing self-embedded variables.

In this section we consider qNSE grammars, namely CFGs in which the only self-embedded variables are unary. As observed by Andrei, Cavadini, and Chin, all the languages generated by these grammars are regular [ACC03]. We are now going to describe and refine the idea used to prove that result, in order to extend Theorem 3.2 to qNSE grammars. First, it is useful to have an upper bound for the cost of the conversion of unary CFGs into equivalent finite automata.

Lemma 3.4. *Each unary* CFG $\mathcal{G}$ *of size* s *can be transformed into an equivalent* NFA *with* $2^{O(s)}$ *states and into an equivalent* DFA *with* $2^{O(s^2)}$ *states.*

Proof. In [PSW02, Thms. 4, 6], it was proved that for any unary CFG in Chomsky normal form with h variables there exists an equivalent NFA with at most $2^{2h-1}+1$ states and, when $h \geq 2$, an equivalent DFA with less than 2^{h^2} states. By inspecting the arguments used in the proof, it can be observed that these bounds do not change if unary CFGs whose production right-hand sides have length at most 2 are considered. Each CFG can be turned in this form with a linear increase of the size. □

Let $\mathcal{G} = \langle V, \Sigma, P, S \rangle$ be a qNSE grammar of size s. We proceed as follows:

- We can suppose that each production right-hand side is either a sequence of variables or a single terminal, *i.e.*, $\alpha \in V^* \cup \Sigma$ for each $A \to \alpha$ in P. This (at most) linearly increases the size of the grammar.

- Let $\mathcal{G}' = \langle V', \Sigma', P', S \rangle$ and $\mathcal{G}'' = \langle V'', \Sigma, P'', S'' \rangle$ be the grammars obtained from $\mathcal{G}$ by choosing as V' the set of variables in $\mathcal{G}$ which generate at least one nonunary terminal string and as $V'' = V \setminus V'$ the set of *unary variables*, namely variables generating unary languages, $\Sigma' = \Sigma \cup V''$, P' is the set of productions in P having left-hand side in V', $P'' = P \setminus P'$, and S'' is a variable in V''. It can be verified that $\mathcal{G} = \mathcal{G}' \oplus \mathcal{G}''$. Furthermore, if the sizes of $\mathcal{G}'$ and $\mathcal{G}''$ are s' and s'', then $s' + s'' = s$. Notice that $\Sigma' = V''$ under the hypothesis that useless variables have been removed from $\mathcal{G}$.

- Since its variables are nonunary, the grammar $\mathcal{G}'$ is NSE. Hence, using Theorem 3.2, there exists an NFA $\mathcal{M}'$ with $2^{O(s')}$ states accepting $L(\mathcal{G}')$.

- For each unary variable $A \in V''$, from the grammar $\mathcal{G}''_{|A}$ we obtain a unary NFA $\mathcal{M}_A$ accepting $L(\mathcal{G}''_{|A})$ with $2^{O(s'')}$ states (Lemma 3.4).

- Finally, we substitute in the automaton $\mathcal{M}'$ the automata $\mathcal{M}_A$, as described in Section 3.1. Hence, we obtain an NFA $\mathcal{M}$ with $2^{O(s')}2^{O(s'')}$ states accepting the language $L(\mathcal{G}) = L(\mathcal{G}' \oplus \mathcal{G}'')$. Since $s' + s'' = s$, the total number of states of $\mathcal{M}$ is $2^{O(s)}$.

From the previous discussion, we obtain the extension of Theorem 3.2 to qNSE grammars.

Theorem 3.4. *Let $\mathcal{G}$ be a* qNSE *grammar of size s. Then there exist an* NFA *and a* DFA *accepting $L(\mathcal{G})$ with $2^{O(s)}$ and $2^{2^{O(s)}}$ many states, respectively.*

The optimality of the bounds in Theorem 3.4 follows from the optimality of those in Theorem 3.2 presented in Section 3.3.

4

Optimal Simulations Between Non-Self-Embedding Grammars, Constant-Height Pushdown Automata, and Limited Automata

In Chapter 3 we investigated the relationships between the description sizes of non-self-embedding grammars and finite automata. In the worst case, the size of a deterministic automaton equivalent to a given non-self-embedding grammar is doubly exponential in the size of the grammar. The gap reduces to a simple exponential in the case of nondeterministic automata.

Other formal models characterizing the class of regular languages and exhibiting gaps of the same order with respect to deterministic and nondeterministic automata have been investigated in the literature. Two of them are constant-height pushdown automata and 1-limited automata. The aim of this chapter is to study the size relationships between non-self-embedding grammars, constant-height pushdown automata, and 1-limited automata, three models that restrict context-free acceptors to the level of regular recognizers.

The exponential and double exponential gaps from constant-height pushdown automata to nondeterministic and deterministic automata have been proved by Geffert, Mereghetti, and Palano [GMP10]. Furthermore, Bednárová et al. showed the interesting result that the gap from nondeterministic to deterministic constant-height pushdown automata is double exponential also [Bed+14]. We remind the reader that both non-self-embedding grammars and constant-height pushdown automata are restrictions of the

corresponding general models, where true recursions are not possible. In the first part of the chapter we compare these two models by proving that they are polynomially related in size.

Also 1-limited automata can be significantly smaller than equivalent finite automata. The equivalence between 1-limited automata and finite automata has been investigated from the descriptional complexity point of view by Pighizzini and Pisoni, who proved exponential and double exponential gaps from 1-limited automata to nondeterministic and deterministic finite automata, respectively [PP14]. Moreover, as we shall see in Chapter 5, two-way nondeterministic finite automata can require exponential size with respect to deterministic 1-LAs even in the unary case.

In the second part of this chapter, we turn our attention to the size relationships between 1-limited automata and non-self-embedding grammars. The main result presented in this chapter is a construction transforming each non-self-embedding grammar into a 1-limited automaton of polynomial size. As a consequence, each constant-height pushdown automaton can be transformed into an equivalent 1-limited automaton of polynomial size. We also prove, using a different construction, that even the conversion of deterministic constant-height pushdown automata into deterministic 1-limited automata costs polynomial in size. For the converse transformation, we show that an exponential size is necessary. Indeed, we prove a stronger result by exhibiting a family of languages $(L_n)_{n=1}^{\infty}$, such that, for each $n \geq 1$, a language L_n accepted by a two-way deterministic finite automaton with $O(n)$ states, which requires exponentially many states to be accepted even by an unrestricted pushdown automaton. From the cost of the conversion of 1-limited automata into nondeterministic automata, it turns out that for the conversion of 1-limited automata into non-self-embedding grammars an exponential size is also sufficient. The results shown in this chapter have been presented in [GPP18] and are summarized in Figure 4.1.

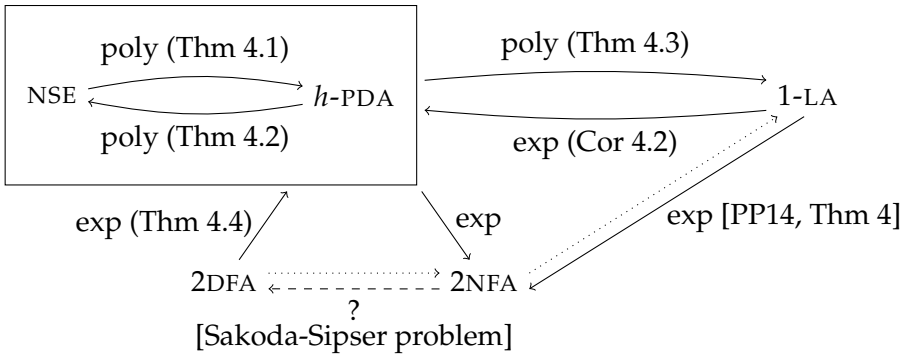

Figure 4.1: Some bounds discussed in the chapter. Dotted arrows denote trivial relationships, while the dashed arrow indicates the Sakoda and Sipser's question [SS78]. The exponential cost of the simulation of h-PDAs by 2NFAs is discussed at the end of Section 4.3.2.

4.1 On the Form of Constant-Height Pushdown Automata

For ease of presentation of the constructions showed in the next sections, following Bednárová et al. [GMP10, Bed+14], we consider PDAs and the restriction of the model in which the capacity of the pushdown store is bounded by some constant $h \in \mathbb{N}$ in the following form, where the transitions manipulating the pushdown store are clearly distinguished from those reading the input tape.

Lemma 4.1. *Every h-PDA $\mathcal{A} = \langle Q, \Sigma, \Gamma, \delta, q_I, Z_0, F \rangle$ can be turned, with a linear increase in size, in a form in which δ is a transition relation $\delta \subseteq Q \times (\{\varepsilon\} \cup \Sigma \cup \{-,+\}\Gamma) \times Q$ with the following meaning:*

- *$(p, \varepsilon, q) \in \delta$: $\mathcal{A}$ can reach the state q from the state p without using the input tape nor the pushdown store (ε-moves);*
- *$(p, a, q) \in \delta$: $\mathcal{A}$ can reach the state q from the state p by reading the symbol a from the input without using the pushdown store;*
- *$(p, -X, q) \in \delta$: if the symbol on the top of the pushdown store is X, $\mathcal{A}$ can reach the state q from the state p by popping off X, not using the input tape;*

- $(p, +X, q) \in \delta$: *if the number of symbols contained in the pushdown store is less than h, $\mathcal{A}$ can reach the state q from the state p by pushing X on the pushdown store, without using the input tape.*

4.2 Non-Self-Embedding Grammars versus Constant-Height Pushdown Automata

In this section we prove that NSE grammars and h-PDAs are polynomially related in size.

4.2.1 From NSE grammars to h-PDAs

The following result is based on the decomposition of NSE grammars recalled in Theorem 3.1.

Studying the relationships between NSE grammars and PDAs, Anselmo, Giammarresi, and Varricchio claimed that from any NSE grammar in *canonical normal form* (namely with productions $A \to a\gamma$ or $A \to \gamma$, $A \in V$, $a \in \Sigma$ and $\gamma \in V^*$), by applying a standard transformation, it is possible to obtain an equivalent constant-height PDA [AGV02]. Unfortunately, the argument fails when the grammar contains left-recursive derivations, *i.e.*, derivations of the form $A \stackrel{*}{\Rightarrow} A\gamma$, with $\gamma \neq \varepsilon$. For them, the resulting PDA has computations with arbitrarily high pushdown stores. This problem can be fixed by replacing each left-linear grammar corresponding to a strongly connected component of the production graph of the given NSE grammar by a set of equivalent right-linear grammars, as shown in the following lemma:

Lemma 4.2. *Each* NSE *grammar can be converted into an equivalent* NSE *grammar of polynomial size which can be expressed as a $\oplus$-composition of right-linear grammars.*

Proof. First of all, we remind the reader that from each left-linear grammar we can obtain an equivalent right-linear grammar whose size is linear in the size of $\mathcal{G}$ (see Lemma 3.2).

Now suppose to have an NSE grammar $\mathcal{G}$, where $\mathcal{G} = \mathcal{G}_1 \oplus \mathcal{G}_2 \oplus \cdots \oplus \mathcal{G}_g$ and each $\mathcal{G}_i$ is left-linear or right-linear. The idea is to define an equivalent grammar $\mathcal{G}'$ by keeping each right-linear $\mathcal{G}_i$, and by replacing each left-linear $\mathcal{G}_i$ by an equivalent right-linear grammar. However, when doing grammar $\oplus$-composition, we could use derivations of $\mathcal{G}_i$ that begin from variables of V_i different than the start symbol S_i. For this reason, we replace each left-linear $\mathcal{G}_i$ by *a family* of right-linear grammars $\mathcal{G}'_{iA}$, with $A \in V_i$, where $\mathcal{G}'_{iA}$ is equivalent to the grammar $\mathcal{G}_{i|A} = \langle V_i, \Sigma_i, P_i, A \rangle$. It can be verified that the size of the resulting grammar $\mathcal{G}'$ is at most quadratic in the size of $\mathcal{G}$. □

We now prove that each NSE grammar can be transformed into an h-PDA of polynomial size.

Theorem 4.1. *Each* NSE *grammar* $\mathcal{G} = \langle V, \Sigma, P, S \rangle$ *can be converted into an* h-PDA $\mathcal{A}$ *with both* h *and the size of* $\mathcal{A}$ *polynomial in the size of* $\mathcal{G}$.

Proof. We start from a NSE grammar $\mathcal{G} = \mathcal{G}_1 \oplus \mathcal{G}_2 \oplus \cdots \oplus \mathcal{G}_g$ such that the sum of the sizes of the $\mathcal{G}_i$'s is polynomial in the size of $\mathcal{G}$, and where each $\mathcal{G}_i = \langle V_i, \Sigma_i, P_i, S_i \rangle$ is right-linear, by Theorem 3.1, Theorem 3.2, and Lemma 4.2. First, as in the construction presented by Anselmo, Giammarresi, and Varricchio, we show that if a variable $A \in V_i$, $1 \leq i \leq g$, derives a string $x\alpha$ by a leftmost derivation, *i.e.*, $A \underset{lm}{\overset{\star}{\Rightarrow}} x\alpha$, where x is the longest prefix of $x\alpha$ consisting only of terminal symbols, then the length of α is linear in $g - i$. More precisely, we claim that $|\alpha| \leq K(g - i) + 1$, where K is the maximum length of production right-hand sides. We are going to prove this claim by induction on the number h of steps of the derivation $A \underset{lm}{\overset{\star}{\Rightarrow}} x\alpha$.

For $h = 0$ we have $|\alpha| = 1$ and, hence, the claim is trivial.

Consider now $h > 0$. If $\alpha = \varepsilon$ then the claim is obvious. Otherwise, let $A \to X_1 X_2 \cdots X_s$ be the first production used in the derivation under consideration, *i.e.*,

$$A \Rightarrow X_1 X_2 \cdots X_s \underset{lm}{\overset{\star}{\Rightarrow}} \alpha_1 \alpha_2 \cdots \alpha_s = x\alpha$$

where $X_k \in \Sigma \cup \bigcup_{j=i}^{g} V_j$ and $X_k \underset{lm}{\overset{\star}{\Rightarrow}} \alpha_k$, for $k = 1, \ldots, s$. Let ℓ, $1 \leq \ell \leq s$, be the smallest

index such that α_ℓ contains at least one variable. Hence, we can write $x = x'x''$ where $x' = \alpha_1\alpha_2\cdots\alpha_{\ell-1}$, $x''\alpha' = \alpha_\ell$, $\alpha_k = X_k$ for $k = \ell+1,\ldots,s$, and $\alpha = \alpha' X_{\ell+1}\cdots X_s$.

Since the derivation $X_\ell \underset{lm}{\overset{\star}{\Rightarrow}} x''\alpha'$ consists of less than h steps, from the induction hypothesis we get that $|\alpha'| \leq K(g-j)+1$, where j is the index satisfying $X_\ell \in V_j$. Thus, $|\alpha| = |\alpha'| + s - \ell \leq K(g-j)+1+s-\ell$.

Due to the fact that $s - \ell < K$, when $j > i$ from the last inequality we obtain $|\alpha| < K(g-i)+1$. Furthermore, since $\mathcal{G}_i$ is right-linear, the case $j = i$ could occur only when $\ell = s$, thus implying $\alpha = \alpha'$ and, hence $|\alpha| = |\alpha'| \leq K(g-i)+1$. This completes the proof of the claim.

From the grammar $\mathcal{G}$ we can apply a standard construction to obtain a PDA $\mathcal{M}$ which simulates a leftmost derivation of $\mathcal{G}$, by replacing any variable A occurring on the top of the pushdown by the right-hand side of a production $A \to \alpha$, and by popping off the pushdown any terminal symbol occurring on the top and matching the next input symbol (for details see, *e.g.*, [HMU01]). After consuming an input prefix y, the pushdown store of $\mathcal{M}$ can contain any string $z\alpha$ such that $S \underset{lm}{\overset{\star}{\Rightarrow}} yz\alpha$, yz is the longest prefix of $yz\alpha$ consisting only of terminal symbols, and z is a suitable factor of the string which was most recently pushed on the pushdown. Since $|z| \leq K$ and, according to the first part of the proof $|\alpha| \leq K(g-1)+1$, we conclude that the pushdown height is bounded by $Kg+1$. Hence, $\mathcal{M}$ is a constant-height PDA. Finally, $\mathcal{M}$ can be converted in the form given in Lemma 4.1 by keeping its size polynomial. □

4.2.2 From h-PDAs to NSE grammars

We first show that, modulo acceptance of the empty word, with only a polynomial increase in the size we can transform any h-PDA in a special form. Subsequently, we shall associate to any h-PDA in such form, a NSE grammar and show that it is equivalent to the h-PDA.

Lemma 4.3. *For each h-PDA $\mathcal{A} = \langle Q, \Sigma, \Gamma, \delta, q_0, F\rangle$ there exists an h-PDA $\mathcal{A}' = \langle Q', \Sigma, \Gamma', \delta', q_-, \{ q_+ \}\rangle$ and a mapping $\widetilde{h} : \Gamma' \to \{ 1,\ldots,h \}$ such that:*

- $L(\mathcal{A}') = L(\mathcal{A}) \setminus \{ \varepsilon \}$;
- *$\mathcal{A}'$ has polynomial size with respect to $\mathcal{A}$;*
- *$\mathcal{A}'$ accepts with empty pushdown;*
- *$\mathcal{A}'$ has no ε-moves;*
- *each symbol $X \in \Gamma'$ can appear on the pushdown only at height $\widetilde{h}(X)$;*
- *every nonempty computation path of $\mathcal{A}'$ starting and ending with the same symbol X on the top of the pushdown, and never popping off X in the meantime, consumes some input letters.*

Proof. First, we create two new states q_- and q_+, intuitively the new initial and unique final states, respectively, and we add transitions (q_-, ε, q_0) and (p, ε, q_+) for each $p \in F$. Furthermore, in order to empty the pushdown store at the end of the accepting computations, we add the transition $(q_+, -X, q_+)$ for each $X \in \Gamma$. We denote by $Q_\star$ the set $Q \cup \{ q_-, q_+ \}$.

Second, by extending $Q_\star$ to $Q_\star \times \{ 0, \ldots, h \}$, we can suppose that each state stores the current height of the pushdown as second component. After this change, we set $(q_-, 0)$ as initial state and $(q_+, 0)$ as unique final state (as a consequence, acceptance is necessarily made with empty pushdown). We then set $\Gamma' = \Gamma \times \{ 1, \ldots, h \}$ and we modify the transitions in such a way that a symbol $(\gamma, i) \in \Gamma'$ can be pushed only from a state in $Q \times \{ i - 1 \}$, *i.e.*, only at pushdown height i. The mapping $\widetilde{h}$ is then defined on Γ' as the projection over the second component.

Now, for each state $(p, i) \in Q_\star \times \{ 0, \ldots, h \}$, we define the set $E_{(p,i)}$ of states (q, i) which are accessible from (p, i) by using only transitions in

$$(Q_\star \times \{ i, \ldots, h \}) \times (\{ \varepsilon \} \cup \{ -, + \}\Gamma) \times (Q_\star \times \{ i, \ldots, h \}).$$

The restriction to states from $Q_\star \times \{ i, \ldots, h \}$ ensures that the considered computation paths can never pop off symbols under their initial level, while the restriction on the set

of actions forbids any reading of the input. We first replace such computations by a single ε-move. This can be achieved as follows:

- we create a transition $((p,i),\varepsilon,(q,i))$ for each $(q,i) \in E_{(p,i)}$;
- we add a new state component storing an element in $\{\,\texttt{push},\texttt{pop},\texttt{read}\,\}$ that saves the last operation performed during the computation (with the natural meaning, ε-moves being not considered as operations) and we forbid transitions of the form $((p,j),-X,(q,j-1))$ whenever the last operation is `push`. (For simplicity, the newly-introduced component will not appear in the end of the proof.)

After such transformation, the only computations that start and end with same symbol on the top of the pushdown, without popping off symbols under the corresponding level, and without scanning any input letter, are necessarily sequences of ε-moves. Hence, each set $E_{(p,i)}$, which is kept unchanged by the above transformation, is now equal to the set of states accessible from (p,i) through a sequence of ε-moves.

We finally proceed to the elimination of ε-moves, using classical techniques. First, we consider the set $E_{(q_-,0)}$ of states that are accessible from the initial configuration through a sequence of ε-moves. For each state $(p,0) \in E_{(q_-,0)}$ and each transition $((p,0),\varkappa,(r,i))$ with $\varkappa \neq \varepsilon$, we create a transition $((q_-,0),\varkappa,(r,i))$. We then remove every ε-move from q_-, *i.e.*, every transition of the form $((q_-,0),\varepsilon,(p,0))$. As a consequence, the empty word cannot be accepted by the resulting h-PDA. However, since every computation of $\mathcal{A}$ accepting a nonempty word should perform a transition of the form $((p,0),\varkappa,(r,i))$ with $\varkappa \neq \varepsilon$ at some point, our transformation preserves acceptance of nonempty words.

Lastly, the remaining ε-moves are eliminated as follows. For each transition of the form $((p,i),\varkappa,(q,j))$ with $\varkappa \neq \varepsilon$ and each $(r,j) \in E_{(q,j)}$, we create the transition $((p,i),\varkappa,(r,j))$. We finally remove all remaining ε-moves.

The complete construction has polynomial cost and the resulting h-PDA $\mathcal{A}'$ accepts an input word if and only if the word is nonempty and was accepted by the original h-PDA $\mathcal{A}$. Acceptance is furthermore done by empty pushdown, indeed the only final state $(q_+,0)$ stores the information that the current pushdown height is 0. Moreover, the

projection $\widetilde{h}$ over the pushdown alphabet Γ', associates to each pushdown symbol, the only height index to which it may appear in the pushdown store. □

By adapting the classical construction of CFGs from PDAs (see, *e.g.*, [HMU01, Sec. 6.3]), from an h-PDA $\mathcal{A} = \langle Q, \Sigma, \Gamma, \delta, q_-, \{ q_+ \} \rangle$ in the form of Lemma 4.3, we define a grammar $\mathcal{G} = \langle V, \Sigma, P, S \rangle$, where V consists of an initial symbol S and of triples of the form $[pXq]$ and $\langle pXq \rangle$, for $q, p \in Q$, $X \in \Gamma_\perp = \Gamma \cup \{ \perp \}$, with the new symbol $\perp \notin \Gamma$ denoting the (missed) bottom of the pushdown store.

Before defining the set P of productions, we give a short explanation of the meaning of the variables we just introduced. Each triple $[pXq]$ is used to generate any string which is consumed in a computation path $\mathcal{C}$ that starts in the state p with X on the top of the stack and ends in the state q with the *same occurrence* of X on the top of the stack, i.e., the height of the stack at the beginning and at the end of $\mathcal{C}$ is the same and it cannot be lower in between. Notice that this implies that $\mathcal{C}$ does not depend on the symbols that are on the stack at the beginning and at the end of $\mathcal{C}$. Any string generated by a variable $\langle pXq \rangle$ is consumed in a computation path $\mathcal{C}$ which, besides the previous conditions, does not visit other configurations with same stack height, namely, either $\mathcal{C}$ consists of a single step, or it starts by pushing a symbol which is popped off only at its last step. As we shall prove, the use of two types of triples allows to obtain an NSE grammar.

We now list the productions in the set P and then we shall prove that the grammar has the desired behavior:

(i) $\langle pXq \rangle \to a$, for $(p, a, q) \in \delta$

(ii) $\langle pXq \rangle \to [p'Yq']$, for $(p, +Y, p'), (q', -Y, q) \in \delta$, *i.e.*, push and pop of a same symbol Y

(iii) $[pXq] \to \langle pXr \rangle [rXq]$, for $p, q, r \in Q$, $X \in \Gamma_\perp$

(iv) $[pXq] \to \langle pXq \rangle$, for $p, q \in Q$, $X \in \Gamma_\perp$

(v) $S \to [q_- \perp q_+]$.

Lemma 4.4. *For each $x \in \Sigma^*$, $p, q \in Q$ and $X \in \Gamma_\perp$, $[pXq] \stackrel{\star}{\Rightarrow} x$ if and only if there exists a computation path $\mathcal{C}$ of $\mathcal{A}$ satisfying the following conditions:*

(1) $\mathcal{C}$ starts in the state p and ends in the state q, in both these configurations the symbol at the top of the pushdown is X;[1]

(2) along $\mathcal{C}$ the pushdown is never popped off under height $\widetilde{h}(X)$.

(3) the input factor consumed along $\mathcal{C}$ is x.

Furthermore, $\langle pXq \rangle \stackrel{\star}{\Rightarrow} x$ if and only if besides the above conditions (1) and (3), the following condition (stronger than (2)) is satisfied:

(2′) in all configurations of $\mathcal{C}$ other than the first and the last one, the pushdown height is greater than $\widetilde{h}(X)$.

Proof. First of all, we are going to prove that for all x, p, q, X as in the statement of the lemma, $[pXq] \stackrel{\star}{\Rightarrow} x$ implies (1), (2), and (3), while $\langle pXq \rangle \stackrel{\star}{\Rightarrow} x$ implies (1), (2′), and (3). We proceed by induction on the length $k \geq 1$ of the derivation $[pXq] \Rightarrow x$ or $\langle pXq \rangle \Rightarrow x$.

For $k = 1$, there are no derivations $[pXq] \Rightarrow x$, while $\langle pXq \rangle \Rightarrow x$ implies that x is a terminal symbol and $(p, x, q) \in \delta$ (the production is of the form (i)), from which (1), (2′), and (3) trivially follow.

Suppose now $k > 1$. The first production applied in a derivation $[pXq] \Rightarrow x$ is either of the form (iii) or of the form (iv). In the first case we have $[pXq] \Rightarrow \langle pXr \rangle [rXq] \Rightarrow x$, $\langle pXr \rangle \Rightarrow x'$, $[rXq] \Rightarrow x''$, for some $r \in Q$, $1 \leq k', k'' < k$, $k' + k'' = k - 1$, $x', x'' \in \Sigma^+$ such that $x'x'' = x$. Using the induction hypothesis, we can find two computations path $\mathcal{C}'$ and $\mathcal{C}''$, from state p to r and from state r to q, respectively, with X at the top of the pushdown at the beginning and at the end, such that the pushdown is never popped under its level at the beginning of these paths, and consuming the factors x' and x'',

[1] When $X = \perp$ the pushdown store in these configurations is empty. Furthermore, we stipulate $\widetilde{h}(\perp) = 0$.

respectively. By concatenating these two paths, we find the path $\mathcal{C}$ satisfying in (1), (2), and (3).

If $[pXq] \Rightarrow \langle pXq\rangle \Rightarrow x$ (*i.e.*, the first production applied is of the form (iv)), (1), (2), and (3) follow from the induction hypothesis applied to the derivation $\langle pXq\rangle \Rightarrow x$.

We now consider a derivation $\langle pXq\rangle \Rightarrow x$, with $k > 1$. The first step can only be of the form (ii), namely $\langle pXq\rangle \Rightarrow [p'Yq'] \Rightarrow x$. From the induction hypothesis, there is a computation path $\mathcal{C}'$ from state p' to state q' which starts and ends with Y at the top of the pushdown, never popping off the pushdown under the initial level, and consuming x from the input tape. From a configuration with state p and X at the top of the pushdown, $\mathcal{A}$ can start a computation path which pushes Y, simulates $\mathcal{C}'$, and finally pops Y off the pushdown. While simulating $\mathcal{C}'$ the pushdown always contains the symbol Y over X. Hence, it is higher than in the first and in the last configuration of $\mathcal{C}$. This proves (1), (2′), and (3).

To prove the converse implications, we proceed by induction on the length k of the computation path $\mathcal{C}$ satisfying conditions (1), (2), (3), and, possibly, the further condition (2′).

If $k = 1$ then $\mathcal{C}$ should consist only of one move, which consumes the input symbol $x = a$ and does not modify the pushdown store. According to the definition of $\mathcal{G}$, the only possible derivations corresponding to such path are $\langle pXq\rangle \Rightarrow x$ and $[pXq] \Rightarrow \langle pXq\rangle \Rightarrow x$.

For $k > 1$ we consider two cases. First we suppose that $\mathcal{C}$ satisfies (1), (2), (3), but does not satisfy (2′). We decompose $\mathcal{C}$ in two shorter paths $\mathcal{C}'$ and $\mathcal{C}''$ that are delimited by the *first configuration* which is reached in $\mathcal{C}$ with the same pushdown height as at the beginning and at the end of $\mathcal{C}$. These two paths satisfy (1), (2), (3). Furthermore, $\mathcal{C}'$ satisfies also (2′). By the induction hypothesis, we get that $\langle pXr\rangle \stackrel{\star}{\Rightarrow} x'$, $[rXq] \stackrel{\star}{\Rightarrow} x''$, where r is the state reached at the end of $\mathcal{C}'$ and $x'x'' = x$. Using production (iv) we obtain the derivation $[pXq] \Rightarrow \langle pXr\rangle[rXq] \stackrel{\star}{\Rightarrow} x'x'' = x$.

If $\mathcal{C}$ satisfies (1), (2′), and (3), then it should start in state p with a push of a symbol Y moving in a state p', ends after a pop of the same symbol Y from a state q' to state q, where

the symbol Y is never popped off the pushdown in between. The path $\mathcal{C}'$, consisting of $k-2$ moves, obtained by removing from $\mathcal{C}$ the first and the last move, consumes the same input string x which is consumed by $\mathcal{C}$. From the induction hypothesis, we obtain that $[p'Yq'] \stackrel{\star}{\Rightarrow} x$ and, considering (ii), $\langle pXq \rangle \Rightarrow [p'Yq'] \stackrel{\star}{\Rightarrow} x$. Furthermore, by (iv), we also obtain $[pXq] \stackrel{\star}{\Rightarrow} x$. □

From Lemma 4.4, considering production (v), we can conclude that the grammar $\mathcal{G}$ so defined is equivalent to the given PDA $\mathcal{A}$.

Lemma 4.5. *The above-defined grammar $\mathcal{G}$ is non-self-embedding.*

Proof. From productions (ii), we observe that $\widetilde{h}(X') > \widetilde{h}(X)$ for any possible variable $\langle p'X'q' \rangle$ or $[p'X'q']$ that can appear in a sentential form from a variable $\langle pXq \rangle$. Hence, each variable $\langle pXq \rangle$ is not self-embedded.

Now, we consider any variable of the form $[pXq]$. We observe that in each derivation $[pXq] \stackrel{+}{\Rightarrow} \alpha[pXq]\beta$, $\alpha, \beta \in (V \cup \Sigma)^*$, the occurrence of $[pXq]$ on the right-hand side can be obtained only if, each time the rightmost variable is rewritten, productions of the form (iii) are used. Hence, the string β must be empty. This allows us to conclude that each $[pXq]$ is not self-embedded. □

By combining the previous results, we obtain:

Theorem 4.2. *For each h-*PDA *there exists an equivalent* NSE *grammar of polynomial size.*

Proof. From Lemmas 4.3, 4.4, and 4.5, from an h-PDA $\mathcal{A}$ we can obtain an NSE grammar $\mathcal{G}$ of polynomial size generating $L(\mathcal{A}) \setminus \{ \varepsilon \}$. In case $\varepsilon \in L(\mathcal{A})$, in order to make $\mathcal{G}$ equivalent to $\mathcal{A}$, we add the production $S \to \varepsilon$. □

As a consequence of Theorems 4.1 and 4.2, by paying a polynomial size increase, each NSE grammar can be transformed into an equivalent one in a particular form.

Corollary 4.1. *Each* NSE *grammar is equivalent (modulo the empty word) to a grammar in Chomsky normal form of polynomial size, in which, for each production $X \to YZ$, Y is greater than X according to the order induced by the production graph.*

Proof. By Theorem 4.1, from each NSE grammar $\mathcal{G}$ we can obtain an equivalent h-PDA $\mathcal{A}$ of polynomial size which, according to Theorem 4.2, can be transformed into an equivalent NSE grammar $\mathcal{G}'$ as defined above. We can observe that if $X \stackrel{+}{\Rightarrow} \alpha X \beta$ in $\mathcal{G}'$, then β should be empty. This implies that for each production $X \to YZ$, Y is greater than X according to the order induced by the production graph. Furthermore, unit productions, namely productions (ii), (iv) and (v), can be easily eliminated, yielding a grammar $\mathcal{G}''$ of the desired form. □

4.3 Non-Self-Embedding Grammars versus 1-Limited Automata

In this section, we compare the sizes of NSE grammars and of h-PDAs with the size of equivalent 1-limited automata. We prove that for each NSE grammar there exists an equivalent 1-LA of polynomial size. As a consequence, the simulation of constant-height PDAs by 1-LAs is polynomial in size.

Concerning the converse transformation, we prove that 1-LAs can be more succinct than NSE grammars and constant-height PDAs. Actually, we prove a stronger result showing the existence of a family of languages $(L_n)_{n=0}^{\infty}$ such that, for each $n > 0$, L_n is accepted by a 2DFA with $O(n)$ states, while each Chomsky normal form grammar or PDA accepting L_n would require an exponential size in n.

4.3.1 From NSE grammars to 1-LAs

We start from an NSE grammar $\mathcal{G} = \langle V, \Sigma, P, S \rangle$ in the form given by Corollary 4.1. Thus, every derivation tree of $\mathcal{G}$ has a particular form which can be expressed using the notion we now introduce. For any constant j, we call *j-tree* any labeled tree satisfying:

- internal nodes are labeled by variables, while leaves are labeled by terminal symbols;

- each internal node either has exactly two children which are internal nodes, or has a unique child which is a leaf;
- the root is an internal node;
- along every branch, the number of *left turns* (*i.e.*, the number of nodes which are left child of some node) is bounded by j.

We observe that any 0-tree consists of a root labeled by some variable which has as unique child a leaf labeled by a terminal. Moreover, a j-tree is also a $(j+1)$-tree. We also point out that we do not require that j-trees are consistent with the production rules of $\mathcal{G}$. However, due to the form given by Corollary 4.1, there exists a constant c such that any derivation tree of $\mathcal{G}$ is a c-tree with root label S.

We now describe how we encode a j-tree T with m leaves into a word u of length m over the alphabet $\Gamma_j = \Sigma \times V \times V \times \{0, \ldots, j\}$. The construction is illustrated in Figure 4.2. First, we inductively index the nodes of T according to the following rules:

- the root of T has index j;
- the left child of a node with index i has index $i-1$;
- the right child of a node with index i has index i;
- a leaf has the same index as its parent.

In other words, the index of a node is an upper limit to the number of left turns from that node to a leaf. From now on, we fix a parameter $Y \in V$ whose meaning will be discussed later. For a leaf ℓ of the j-tree labeled by a symbol $a \in \Sigma$, we consider the symbol $\sigma_\ell = \langle a, X, Z, i \rangle \in \Gamma_j$ where (X, i) is the indexed label of the closest ancestor of ℓ which is not a right child of any node (such nodes have square shape in Figure 4.2) and Z is the label of its right sibling if any, or equals Y otherwise, namely when that node is the root of the tree. Intuitively, the *j-compression* of the j-tree T is the word $\sigma_{\ell_1} \cdots \sigma_{\ell_m}$ where $\ell_1, \ldots, \ell_m$ are the leaves of T taken from left to right. Formally, it is inductively

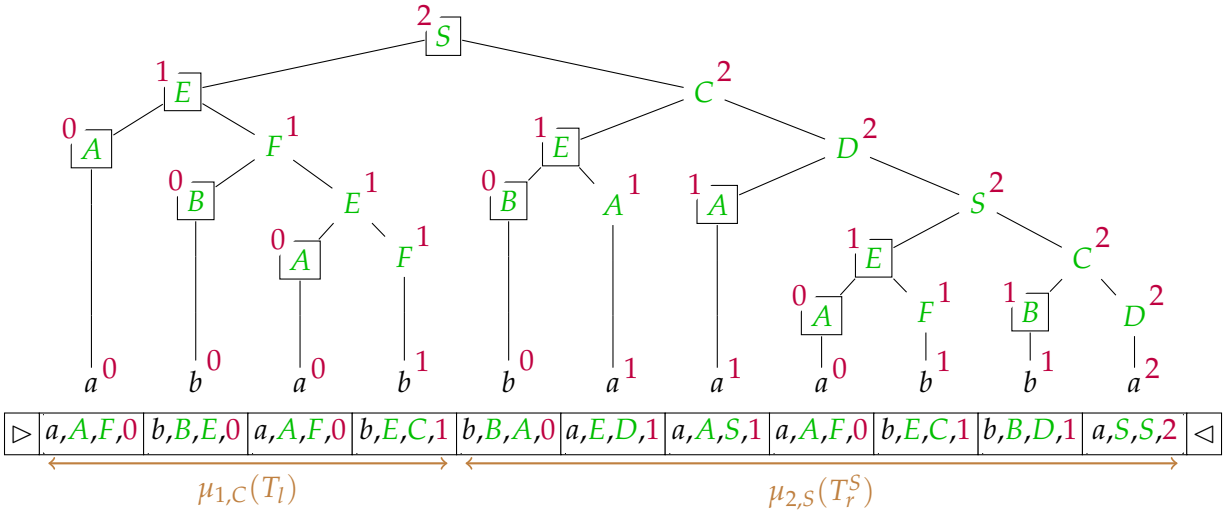

Figure 4.2: An example of a 2-tree and its 2-compression. Here T_l and T_r denote the left and right subtrees at depth 1 and T_r^S denotes the subtree T_r in which the label of the root has been changed to S.

defined as follows. Let $Y \in V$ and T be a j-tree with root label X, for some $j \geq 0$. If T consists of its root with only one child being a leaf labeled by $a \in \Sigma$ (which is always the case when $j = 0$ by definition of j-trees), then $\mu_{j,Y}(T) = \langle a, X, Y, j \rangle$. Otherwise, $j > 0$ and T consists of a root node yielding a left subtree T_l and a right subtree T_r. Let Z be the root label of T_r. Then, denoting T_r^X the tree T_r in which the label of the root has been changed to X, $\mu_{j,Y}(T) = \mu_{j-1,Z}(T_l) \cdot \mu_{j,Y}(T_r^X)$ (see Figure 4.2).

Let us shortly discuss the parameter Y, which does not influence much the j-compression: it occurs only in the rightmost symbol of the word encoding the given j-tree. The parameter is meant to indicate the right sibling label of the root node, when considering left subtrees of some $(j-1)$-tree. Hence, when considering full derivation trees of $\mathcal{G}$, this parameter is meaningless, and we can fix it to some arbitrarily chosen variable, say S (as in Figure 4.2). So defined, the c-compression (or simply *compression*) of a derivation tree of G is a word $u \in \Gamma_c^+$ whose projection on Σ^* is the word generated by the tree. The following remark, which directly follows from the inductive definition of $\mu_{j,Y}$, is instrumental for our later proofs.

Remark 4.1. *If $\mu_{j,Y}(T) = v \cdot \langle a, Z, Y, j\rangle$ then $\mu_{j,Y}(T^X) = v \cdot \langle a, X, Y, j\rangle$, where T^X denotes the tree T in which the root label has been changed to X.*

Not every word over Γ_j is the j-compression of a j-tree, and not every j-tree is a derivation tree. We now introduce a property which allows us to check that a word $u \in \Gamma_j^*$ is a correct j-compression and that the tree it encodes is a derivation tree, namely it is consistent with the production rules of $\mathcal{G}$. We set $\Gamma_{-1} = \emptyset$. A word $u \in \Gamma_j^+$ is a *valid j-compression*, $0 \leq j \leq c$, if, on the one hand, $u = w \cdot \langle a, X, Y, j\rangle$ for some $w \in \Gamma_{j-1}^*$, $a \in \Sigma$ and $X, Y \in V$, and, on the other hand, one of the following two cases holds:

1. $w = \varepsilon$, and $X \to a$ belongs to P;
2. there exist $v, w' \in \Gamma_{j-1}^*$, $b \in \Sigma$, and $W, Z \in V$ such that:
   (a) $w = v \cdot \langle b, W, Z, j-1\rangle w'$
   (b) $X \to WZ$ belongs to P;
   (c) $v \cdot \langle b, W, Z, j-1\rangle$ is a valid $(j-1)$-compression;
   (d) $w' \cdot \langle a, Z, Y, j\rangle$ is a valid j-compression.

In particular, valid 0-compressions are exactly the single-letter words $\langle a, X, Y, 0\rangle$ such that $X \to a \in P$. Observe that Item 2c implies $v \in \Gamma_{j-2}^*$ and therefore, the decomposition of w (Item 2a) as well as W, Z, and b are determined by the leftmost symbol of index $j-1$ of u. Notice furthermore that validity does not depend on the variable Y.

Intuitively, validity of compressions corresponds to derivation consistency of encoded trees. This is stated formally in the following lemma (remember that $\mathcal{G}_{|X}$ denotes the grammar $\mathcal{G}$ in which the starting symbol has been replaced by X).

Lemma 4.6. *Let $j \in \{0, \ldots, c\}$, $X, Y \in V$, $a \in \Sigma$, and $u \in \Gamma_{j-1}^* \cdot \langle a, X, Y, j\rangle$. Then u is a valid j-compression if and only if $u = \mu_{j,Y}(T)$ for some derivation tree T of $\mathcal{G}_{|X}$. In particular, the projection w of u to Σ^* is generated by $\mathcal{G}_{|X}$ through T.*

Proof. We fix a valid j-compression $u \in \Gamma_{j-1}^* \cdot \langle a, X, Y, j\rangle$. We show, by induction on the length of u, that there exists a derivation tree T of $\mathcal{G}_{|X}$ such that $\mu_{j,Y}(T) = u$.

If $|u| = 1$, then $u = \langle a, X, Y, j\rangle$ and $X \to a \in P$ by Item 1. Hence, $u = \mu_{j,Y}(T)$ for T the derivation tree of $\mathcal{G}_{|X}$ which derives the word a from X in one step.

Otherwise, $u = v \cdot \langle b, W, Z, j-1\rangle \cdot w' \cdot \langle a, X, Y, j\rangle$ by Item 2a, where $X \to WZ$ by Item 2b. Moreover, on the one hand $v \cdot \langle b, W, Z, j-1\rangle$ is a valid $(j-1)$-compression by Item 2c, on the other hand $w' \cdot \langle a, Z, Y, j\rangle$ is a valid j-compression by Item 2d. By induction, there exist two derivation trees T_l and T_r, respectively of $\mathcal{G}_{|W}$ and $\mathcal{G}_{|Z}$, such that $\mu_{j-1,Z}(T_l) = v \cdot \langle b, W, Z, j-1\rangle$ and $\mu_{j,Y}(T_r) = w' \cdot \langle a, Z, Y, j\rangle$. Since changing the label of the root node does affect only the rightmost symbol of its compression (Remark 4.1), we have $\mu_{j,Y}(T_r^X) = w' \cdot \langle a, X, Y, j\rangle$ where T_r^X is the tree T_r in which the root label has been changed to X. Consider the tree T consisting of a root labeled by X which has T_l as left subtree and T_r as right subtree. By the above properties, T is a derivation tree of $\mathcal{G}_{|X}$. Moreover, $\mu_{j,Y}(T) = \mu_{j-1,Z}(T_l) \cdot \mu_{j,Y}(T_r^X) = u$.

Conversely, we fix a derivation tree T of $\mathcal{G}_{|X}$ that we supposed to be a j-tree. We show by induction on the structure of T, that $\mu_{j,Y}(T)$ is valid for any $Y \in V$.

If T is a trivial derivation tree consisting of the root node which has as unique child a leaf labeled by a, then, $X \to a \in P$ by definition, whence $\mu_{j,Y}(T) = \langle a, X, Y, j\rangle$ is valid through Item 1 for any Y and any j.

Otherwise, the root of T has two children, yielding a left subtree T_l and a right subtree T_r. Let W and Z be the respective root labels of T_l and T_r. By definition, $u = \mu_{j,Y}(T) = \mu_{j-1,Z}(T_l) \cdot \mu_{j,Y}(T_r^X)$ where T_r^X denotes the tree T_r in which the root label has been changed to X. Since T is a derivation tree, we have $X \to WZ \in P$ (*i.e.*, Item 2b). By induction, $\mu_{j-1,Z}(T_l) = v \cdot \langle b, W, Z, j-1\rangle$ is a valid $(j-1)$-compression (Item 2c), and $\mu_{j,Y}(T_r) = w' \cdot \langle a, Z, Y, j\rangle$ is a valid j-compression (Item 2d). Finally, since modifying the root label of a tree does only affect the rightmost letter of its compressions, by Remark 4.1, we obtain that $u = v \cdot \langle b, W, Z, j-1\rangle w' \cdot \langle a, X, Y, j\rangle$ (*i.e.*, Item 2a). □

Lemma 4.6 yields a strategy to check whether a word $w \in \Sigma^+$ is generated by $\mathcal{G}$, using the property that all its derivation trees are c-trees. We first guess the c-compression of a derivation tree generating w, thus obtaining a word $u \in \Gamma_c^+$ whose projection to Σ

equals w. We then check that u is a valid c-compression with parameter $Y = S$. Although the initial guess makes use of nondeterminism, the verification can be performed deterministically once the guessed symbols have been fixed. We now show how to do this verification with a 2DFA. From now on, we set $\Gamma = \Gamma_c$.

In any valid j-compression of length greater than 1, some factors represent the $(j-1)$-compressions of some subtrees of the encoded tree. They are exactly the factors delimited to the left by a symbol of index greater than or equal to j or by the left endmarker (not included in the factor), and to the right by the symbol of index j corresponding to its root node (included in the factor). In other words, they are maximal factors in $\Gamma_{j-1}^* \cdot \Gamma_j$. This allows a reading head to locally detect the boundaries of such factors when scanning the j-compression. This also implies that the index of a symbol preceding a symbol of index j is always less than or equal to $j-1$. For instance, the compression illustrated in Figure 4.2 admits five valid 1-compression factors, namely the factor $\langle a, A, F, 0\rangle\langle b, B, E, 0\rangle\langle a, A, F, 0\rangle\langle b, E, C, 1\rangle$, the factor $\langle b, B, A, 0\rangle\langle a, E, D, 1\rangle$, the factor $\langle a, A, S, 1\rangle$, the factor $\langle a, A, F, 0\rangle\langle b, E, C, 1\rangle$, and the factor $\langle b, B, D, 1\rangle$ which respectively correspond to the five subtrees rooted in the square-shape nodes which have index 1.

We now describe how a 2DFA $\mathcal{A}$ can check that a word $u \in \Gamma_c^+$ is a valid c-compression. First of all, the device checks that u belongs to $\Gamma_{c-1}^* \cdot \langle a, S, S, c\rangle$ for some letter $a \in \Sigma$. Then, it iteratively verifies that every maximal factor of the form $\Gamma_{j-1}^* \cdot \langle a, X, Y, j\rangle$ is a valid j-compression. To this end, once the verification has been performed for the level $j-1$, it just needs to check that the letter just before $\langle a, X, Y, j\rangle$, if any, is of index at most $j-1$, and that there is consistency between letters of index $j-1$ and $\langle a, X, Y, j\rangle$ of such maximal factor, as follows: sweeping these letters $\langle a_1, W_1, Z_1, j-1\rangle, \ldots, \langle a_k, W_k, Z_k, j-1\rangle$ from left to right and setting $Z_0 = X$, the 2DFA sequentially checks that $Z_{i-1} \to W_i Z_i \in P$ for $i = 1, \ldots, k$, and $Z_k \to a \in P$. In other words, the device implements the above-given inductive definition of valid compressions, with the difference that it tests each subtree of level from 0 to c instead of performing recursive calls. This allows to store only one

Procedure 1: `CheckTree`

```
                                  /* start with the head on the left endmarker */
1 CheckRoot
2 for j ← 0 to c do
3     repeat move the head to the right until index(σ) ≥ j
4     while index(σ) = j do
5         CheckSubtree(j)
6         repeat move the head to the right until index(σ) ≥ j
7     repeat move the head to the left until σ = ▷
8 ACCEPT
```

variable Z at each time.

The 2DFA $\mathcal{A}$ implements a collection of deterministic subroutines, the top-level of which is the procedure `CheckTree`. In each subroutine, σ denotes the symbol currently scanned by the head, which is automatically updated at each head move. Moreover, the special instruction REJECT causes the whole computation to halt and reject. We furthermore use the four natural projections over Γ: for a symbol $\sigma = \langle a, X, Y, j \rangle \in \Gamma$, we set $\texttt{letter}(\sigma) = a$, $\texttt{varLeft}(\sigma) = X$, $\texttt{varRight}(\sigma) = Y$, and $\texttt{index}(\sigma) = j$. We fix the convention $\texttt{index}(\triangleright) = \texttt{index}(\triangleleft) = c + 1$.

Procedure 2: `CheckRoot`

```
9  repeat move the head to the right until σ = ◁
10 move the head to the left
11 if varLeft(σ) ≠ S or varRight(σ) ≠ S or index(σ) ≠ c then REJECT
12 while σ ≠ ▷ do
13     move the head to the left
14     if index(σ) = c then REJECT
```

As initial phase, the subroutine `CheckRoot` checks that the input word belongs to $\Gamma_{c-1}^* \cdot$

Procedure 3: `CheckSubtree(`j`)`

```
                    /* start with the head scanning a symbol of index j */
15 C ← varLeft(σ)
16 repeat move the head to the left until index(σ) ≥ j
17 SelectNext(j − 1)
18 while index(σ) ≠ j do
19 |   if C → varLeft(σ)varRight(σ) ∉ P then REJECT
20 |   C ← varRight(σ)
21 |   SelectNext(j − 1)
22 if C → letter(σ) ∉ P then REJECT
```

Procedure 4: `SelectNext(`j`)`

```
23 move the head to the right
24 if index(σ) ≠ j + 1 then
25 |   while index(σ) < j do move the head to the right
26 |   if index(σ) ≠ j then REJECT
```

$\langle a, S, S, c\rangle$ for some letter $a \in \Sigma$. Then, $\mathcal{A}$ checks the validity of each compression of each level from 0 to c (Lines 2 to 7). This verification uses the procedure `CheckSubtree` (Line 5).

This latter subroutine is the direct implementation of the inductive definition of valid compressions, where the recursive call to incremented level (Item 2c) is omitted (the validity of these sub-compressions have already been checked by previous call to `CheckSubtree`). It uses the subroutine `SelectNext` to locate the leftmost symbol of index $j-1$ in the factor under consideration, if any, or to check if the factor has length 1, otherwise, thus checking Item 2a (or, partially, Item 1). Items 1 and 2b correspond to Lines 22 and 19, respectively, where C contains the variable Z (Line 20), the variable label of the root of the subtree which is initially set to X (Line 15), thus allowing to verify Item 2d (Lines 18 to 21).

To summarize, we obtained the following result.

Lemma 4.7. *The language of valid compressions of derivation trees of $\mathcal{G}$ is recognized by a* 2DFA *which uses $O(c \cdot \#V)$ states.*

Proof. The construction of such a 2DFA $\mathcal{A}$ has been described above. We now estimate its size. The only memory used in the procedure `CheckTree`, is an index $j \in \{0, \dots, c\}$, to which both the subroutines `SelectNext` and `CheckSubtree` have read-only access. The procedure `CheckSubtree` stores one variable, C, which ranges over V. The subroutines `CheckRoot` and `SelectNext` use no additional memory. Hence, the number of states of $\mathcal{A}$ is linear in $c \cdot \#V$. □

We are now ready to state our main result.

Theorem 4.3. *Every* NSE *grammar $\mathcal{G}$ can be transformed into a* 1-LA *$\mathcal{A}$ whose size is polynomial in the size of $\mathcal{G}$.*

Proof. From an NSE grammar $\mathcal{G}$, we obtain an NSE grammar $\mathcal{G}'$ over Σ of polynomial size in the form given by Corollary 4.1, such that $L(\mathcal{G}') = L(\mathcal{G}) \setminus \{\varepsilon\}$.

The automaton $\mathcal{A}$ first performs a left-to-right traversal during which rewrites each letter $a \in \Sigma$ with a symbol $\langle a, X, Y, j \rangle$ for some variables X and Y of $\mathcal{G}'$ and some index $j \leq \#V$. Then it deterministically simulates the behavior of a 2DFA, using Lemma 4.7, which accepts if and only if the contents of the tape before the first traversal was generated by $\mathcal{G}'$, by Lemma 4.6. In case $\varepsilon \in L(\mathcal{G})$, we modify $\mathcal{A}$ in order to accept ε. □

4.3.2 From 1-LAs to NSE grammars: An exponential gap

In this section, we exhibit an infinite family $(L_n)_{n=0}^{\infty}$ of languages over the alphabet $\{0, 1\}$, such that each L_n is recognized by a 1-LA with size polynomial in n, but requires an exponential size in order to be recognized by any h-PDA or NSE grammars. We can actually prove a stronger result, since each L_n is recognized by a 2DFA (and even by a *rotating* deterministic automaton, in which all passes over the input are from left to right [KKM12]) of linear size, while any grammar in Chomsky normal form generating L_n requires an exponential number of variables. As a consequence, every PDA recognizing L_n requires

an exponential size. The proof of this lower bound is obtained by using the *interchange lemma for context-free languages* [ORW85]:

Lemma 4.8. *Let $\mathcal{G}$ be a context-free grammar in Chomsky normal form, with c variables, and let L be the language it generates. For all integers N, m, with $2 \leq m \leq N$, and all subsets R of $L \cap \Sigma^N$, there exists a subset $Z \subseteq R$ with $Z = \{ z_1, z_2, \ldots, z_k \}$ such that $k \geq \frac{\#R}{cN^2}$, and there exist decompositions $z_i = w_i x_i y_i$, with $1 \leq i \leq k$, such that the following conditions are satisfied:*

1. $|w_1| = |w_2| = \cdots = |w_k|$;
2. $|y_1| = |y_2| = \cdots = |y_k|$;
3. $\frac{m}{2} < |x_1| = |x_2| = \cdots = |x_k| \leq m$;
4. $w_i x_j y_i \in L$ *for all* i, j *with* $1 \leq i, j \leq k$.

Theorem 4.4. *For each $n > 0$, let L_n be the language $\{ uuu \mid u \in \{ 0,1 \}^n \}$. Then:*

- *L_n is accepted by a* 2DFA *of size $O(n)$;*
- *each context-free grammar in Chomsky normal form needs exponentially many variables in n to generate L_n;*
- *the size of any* PDA *accepting L_n is at least exponential in n.*

Proof. A 2DFA $\mathcal{A}$ with $O(n)$ states can accept L_n as follows. First $\mathcal{A}$ traverses the whole input tape, in order to verify that the input length is $3n$. Then $\mathcal{A}$, by moving the head back and forth, verifies that all two symbols at distance n are equal. It is not difficult to observe that $\mathcal{A}$ can be implemented using $O(n)$ states.

To prove that each context-free grammar generating L_n requires an exponential number of variables, we observe that given $u, u' \in \{ 0,1 \}^n$, if we decompose the strings $z = uuu$ and $z' = u'u'u' \in L_n$ as $z = wxy$ and $z' = w'x'y'$, with $|w| = |w'|$, $|y| = |y'|$, $n < |x| = |x'| \leq 2n$, then $|wy| \geq n$, thus implying that $u = u_l u_r$ where u_l is a prefix of w and u_r is a suffix of y. If $u \neq u'$ then $x \neq x'$ and the string $wx'y$ cannot belong to L_n.

Applying Lemma 4.8, with $N = 3n$, $R = L_n$ and $m = 2n$, from the previous argument it follows that the resulting set Z cannot contain more than one string. Hence, we conclude that each context-free grammar in Chomsky normal form generating L_n should have at least $\frac{2^n}{9n^2}$ variables.

Finally, since each PDA can be converted into an equivalent context-free grammar in Chomsky normal form with a polynomial number of variables, *e.g.*, [PSW02, Theorem 8] we conclude that the size of any PDA accepting L_n is at least exponential in n. □

Corollary 4.2. *The size cost of the conversion of* 1-LAs *into* NSE *grammars and* h-PDAs *is exponential.*

Proof. The lower bound derives from Theorem 4.4. For the upper bound, in [PP14] it was proved that each 1-LA can be transformed into a 1NFA of exponential size from which, by a standard construction, we can obtain a regular (and, so, NSE) grammar, without increasing the size asymptotically. □

In [Bed+14], the question of the cost of the conversion of deterministic h-PDAs into 1NFAs was raised. To this regard, we observe that the language $(a^{2^n})^*$ is accepted by a deterministic h-PDA of size polynomial in n for large enough h (see, *e.g.*, [Pig09a]) but, by a standard pumping argument, it requires at least 2^n states to be accepted by 1NFAs. Actually, as a consequence of state lower bound presented in [MP00], 2^n states are also necessary to accept it on each 2NFA. Considering Theorem 4.4, we can conclude that both simulations from two-way automata to h-PDAs and from h-PDAs to two-way automata cost at least exponential.

4.4 Deterministic Constant-Height Pushdown Automata versus Deterministic 1-Limited Automata

From the results in Sections 4.2 and 4.3, it turns out that there is a polynomial-size conversion of h-PDAs into 1-LAs. Here, we consider the *deterministic case*. We present a poly-

nomial size conversion of h-DPDAs into deterministic 1-LAs.

We now recall the definition of h-DPDAs, according to [Bed+12]. Let $\mathcal{A} = \langle Q, \Sigma, \Gamma, \delta, q_0, F \rangle$ be an h-PDA. Let Q_Σ, Q_+, and Q_- be the sets of states p such that there exists a transition $(p, \mathtt{op}, q)$ with $\mathtt{op}$ belonging to Σ, to $\{+\}\Gamma$, or to $\{-\}\Gamma$, respectively. $\mathcal{A}$ is *deterministic* if it satisfies the following properties:

1. it does not allow transitions of the form (p, ε, q);
2. Q_Σ, Q_+, and Q_- form a partition of Q;
3. if $(p, \mathtt{op}, q)$ and $(p, \mathtt{op}', q')$ are distinct transitions, then $p \in Q_\Sigma \cup Q_-$ and $\mathtt{op}$ and $\mathtt{op}'$ are distinct elements of Σ if $p \in Q_\Sigma$, or of $\{-\}\Gamma$ if $p \in Q_-$ (notice that this implies that there exists exactly one outgoing transition from each state in Q_+);
4. $F \subseteq Q_\Sigma$.

Item 2 ensures that the action to perform is fully determined by the current state. Based on this, Item 3 states that for any configuration there exists at most one outgoing transition, while Item 4 constrains acceptance, as explained in the following. As for nondeterministic h-PDA, the machine accepts the input word if it reaches an accepting state after having read all the input symbols. However, in order to avoid exiting an accepting configuration, Item 4 requires that the machine halts by waiting for a next symbol to scan. We point out that, in the definition of h-DPDA given in [Bed+12], states with no outgoing transitions are present and transitions of the form (p, ε, q) are allowed under the restriction that from each state at most one such transition is allowed, which should further be the unique transition outgoing that state. With classical transformations, states without any outgoing transition as well as ε-moves can be avoided without increasing the size of the automaton, while preserving determinism. Hence Item 1 as well as the statement of Item 2 which is stronger than those given in [Bed+12] can be ensured without loss of generality.

Moreover, given a h-DPDA, it is always possible to obtain an equivalent h-DPDA of at most the same size, in which there are no push transitions entering a state of Q_-. Indeed, in any computation, if a pop $(p_-, -Y, q)$ immediately follows a push $(p_+, +X, p_-)$

then $X = Y$. Thus, in presence of a transition $(p_+, +X, p_-)$ with $p_- \in Q_-$, we can eliminate the state p_+ and its outgoing transition after modifying the machine as follows: if there exists q such that $(p_-, -X, q) \in \delta$ then we replace each transition $(p, \mathtt{op}, p_+)$ with the transition $(p, \mathtt{op}, q)$ and, furthermore, q becomes the initial state whenever p_+ is initial.[2] By iteratively applying this transformation, an equivalent h-DPDA without any transition in $Q_+ \times \{+\}\Gamma \times Q_-$ is obtained. Finally, we can assume without loss of generality that h-DPDAs do not contain any loop composed by push transitions only. Indeed, without increasing the size of the model, these loops can be eliminated since when entering such a loop, the machine surely halts and rejects after at most h steps as the pushdown height will exceed its bound.

From now on, we assume without loss of generality that h-DPDAs have neither a transition in $Q_+ \times \{+\}\Gamma \times Q_-$ nor a loop with push transitions only.

Let $\mathcal{A} = \langle Q, \Sigma, \Gamma, \delta, q_0, F \rangle$ be a h-DPDA, with Q_Σ, Q_+, Q_- defined as above. By determinism, a state from Q_+ has a unique outgoing transition. Hence, until leaving Q_+, the successive transitions from a state $p \in Q_+$ generate a finite sequence of push transitions, which is fully determined from p. For ease of presentation we shall use the following functions that are defined for each state from Q_+:

- $\eta : Q_+ \to Q_\Sigma$ maps each state in Q_+ to the first reachable state not belonging to Q_+ after a sequence of push transitions, which exists and belongs to Q_Σ by assumption.
- $\ell : Q_+ \to \{1, \ldots, h\}$ maps each state $p_+ \in Q_+$ to the maximum number of consecutive push transitions that can be performed starting from p_+.
- $\omega : Q_+ \to \Gamma^{\leq h}$ maps each state $p_+ \in Q_+$ to the string that can be pushed during a maximal sequence of consecutive push transitions starting from p_+. Notice that the length of such a string is given by $\ell(p_+) \leq h$.

[2] If p_+ is the initial state but there exists no q such that $(p_-, -X, q) \in \delta$, then the complete device recognizes the empty language, and can thus be replaced by a simpler single-state one.

For instance, consider the maximal sequence of consecutive push transitions

$$(p_0, +X_1, p_1), (p_1, +X_2, p_2), \ldots, (p_{n-1}, +X_n, p_n),$$

where $p_n \in Q_\Sigma$ and, for each i such that $0 \leq i < n \leq h$, $p_i \in Q_+$ and $X_{i+1} \in \Gamma$. Then, $\eta(p_0) = p_n$, $\ell(p_0) = n$, and $\omega(p_0) = X_1 \cdots X_n$. These functions can be always computed by analyzing the transition function δ.

Let us now show how the simulation works.

Theorem 4.5. *Each h-*DPDA *admits an equivalent deterministic* 1-LA *of polynomial size.*

Proof. Let $\mathcal{A} = \langle Q, \Sigma, \Gamma, \delta, q_0, F \rangle$ be a h-DPDA where, using the above notations, $Q = Q_\Sigma \cup Q_+ \cup Q_-$. Hence, in every accepting computation, a sequence of push transitions ends by entering a state from Q_Σ from which $\mathcal{A}$ scans the next input symbol, or accepts. We define a simulating deterministic 1-LA $\mathcal{A}' = \langle Q', \Sigma, \Gamma', \delta', q_0', F' \rangle$.

The main idea of the simulation is that at each step of its computation $\mathcal{A}'$ is able to recover the content $w \in \Gamma^{\leq h}$ of the pushdown store of the simulated machine without storing w in its finite control, but from the information written on the already visited tape cells. The simulation, detailed below, is described in Procedures 5, 6, and 7.

More precisely, we assume that $\mathcal{A}'$ stores in its finite control the current pushdown height, in a variable height with maximum value h that is initially set to 0, and the current state, in a variable simulatedState initially containing the initial state of $\mathcal{A}$, of the configuration reached in the simulated computation of $\mathcal{A}$. In order to simulate a maximal sequence of push transitions

$$(p_0, +X_1, p_1), (p_1, +X_2, p_2), \ldots, (p_{n-1}, +X_n, p_n)$$

where $p_0, \ldots, p_{n-1} \in Q_+$, $X_1, \ldots, X_n \in \Gamma$, and $p_n \in Q_\Sigma$, $\mathcal{A}'$ moves its head rightward to the leftmost tape cell which has not been visited so far (*i.e.*, which has not been rewritten) and performs, in one step, the following actions:

- it scans the input symbol $a \in \Sigma$, and determines the transition (p_n, a, q) (Line 30);

Procedure 5: `simulatePush` is called when a push has to be simulated, *i.e.*, when the variable simulatedState contains a state $q \in Q_+$

27 move the head rightward until reaching the leftmost symbol $\sigma \in \Sigma \cup \{ \lhd \}$
28 **if** $\sigma = \lhd$ **then**
29 **if** $\eta(q) \in F$ **then** ACCEPT **else** REJECT
30 detect move $(\eta(q), \sigma, r)$ to be simulated
31 `write(`$\langle q, \mathsf{height} \rangle$`)`
32 **if** $\mathsf{height} + \ell(q) > h$ **then** REJECT
33 $\mathsf{height} \leftarrow \mathsf{height} + \ell(q)$
34 $\mathsf{simulatedState} \leftarrow r$

- it overwrites the cell content with the pair (p_0, i) where $i \leq h - \ell(p_0)$ is the current pushdown height, stored in its finite control (Line 31);
- if the height of the stack after pushing $\ell(p_0)$ symbols does not exceed h (Line 32), it updates the pushdown height component to $i + \ell(p_0) \leq h$ (Line 33);
- it updates the state component to q (Line 34).

Hence, $\mathcal{A}'$ does not only simulate the sequence of push, but also the successive scan step. When all the cells have already been visited, *i.e.*, when the head of $\mathcal{A}'$ has reached the right endmarker, then it halts and accepts if and only if $p_n \in F$ (Line 29).

When the next transition to be simulated has to scan the input (not just after a sequence of push), $\mathcal{A}'$ proceeds similarly (Procedure 6), but simply updates the variable simulatedState accordingly (Line 40) without modifying the value of height, and rewrites the corresponding cell content with the special symbol $\sharp$ (Line 39).

When $\mathcal{A}'$ has to access the content of the pushdown, namely when a pop transition has to be simulated, the simulating machine looks for the last sequence of simulated push transitions whose first symbol was pushed from a level lower than or equal to the current pushdown height. This can be done by scanning leftward the tape, until reaching

Procedure 6: `simulateRead` is called when a read has to be simulated, *i.e.*, when the variable simulatedState contains a state $q \in Q_\Sigma$

35 move the head rightward until reaching the leftmost symbol $\sigma \in \Sigma \cup \{ \lhd \}$
36 **if** $\sigma = \lhd$ **then**
37 **if** $\eta(q) \in F$ **then** ACCEPT **else** REJECT
38 detect move (q, σ, r) to be simulated
39 `write(♯)`
40 simulatedState $\leftarrow r$

Procedure 7: `simulatePop` is called when a pop has to be simulated, *i.e.*, when the variable simulatedState contains a state $q \in Q_-$.

41 move the head leftward until reaching a symbol (p_+, i) with $i \leq$ height
42 detect move $(q, -X, r)$ to be simulated where X is the (height $- i$)-th symbol of $\omega(p_+)$
43 height $\leftarrow$ height $- 1$
44 simulatedState $\leftarrow r$

the rightmost cell containing a pair (p_+, i), with i less than or equal to the value of height (Line 41). At this point, by using i and $\omega(p_+)$, $\mathcal{A}'$ recovers the symbol at level equal to the height stored in the finite control and detects a suitable pop transition to be simulated (Line 42). If such a transition exists then $\mathcal{A}'$ updates both its state and pushdown height components according to this transition (Lines 44, and 43), and continues the simulation. Notice that $\mathcal{A}'$ does not need to recover the original head position, *i.e.*, the head position of $\mathcal{A}$ in the simulated computation, until it enters a state from $Q_\Sigma \cup Q_+$. When this happens, $\mathcal{A}'$ proceed as explained previously.

If no move in δ can be simulated because no suitable transition is defined (Lines 30, 38, and 42), then the simulating machine halts and rejects.

We now evaluate the size of the simulating deterministic 1-LA $\mathcal{A}'$. Notice that the

quantities computed by η, ℓ, and ω do not depend on the input, whence are pre-computed and hardly encoded in the transition table of $\mathcal{A}'$. The working alphabet of $\mathcal{A}'$ is included in $Q_+ \times \{ 0, \ldots, h-1 \} \cup \{ \sharp \}$, while the finite control stores two variables: one state in Q and one pushdown height in $\{ 0, \ldots, h \}$. Hence, the size of $\mathcal{A}'$ is polynomial in $\#Q$ and h. We point out that it does not depend on the size of the pushdown alphabet of $\mathcal{A}$. □

5

Unary Limited Automata

In this chapter we focus on limited automata whose input alphabet is composed by just one symbol.

Limited automata have been investigated from the descriptional complexity point of view by Pighizzini and Pisoni, who proved that each 1-limited automaton $\mathcal{A}$ with n states can be simulated by a one-way deterministic automaton with a number of states double exponential in a polynomial in n. Furthermore, in the worst case, double exponentially many states are necessary for this simulation. The cost reduces to a single exponential when $\mathcal{A}$ is deterministic [PP14].

Theorem 5.1 ([PP14])**.** *Let $\mathcal{M}$ be an n-state* 1-LA*. Then $\mathcal{M}$ can be simulated by a* 1NFA *with $n \cdot 2^{n^2}$ states and by a* 1DFA *with $2^{n \cdot 2^{n^2}}$ states. Furthermore, if $\mathcal{M}$ is deterministic then an equivalent* 1DFA *with no more than $n \cdot (n+1)^n$ states can be obtained.*

The lower bounds in this result have been obtained by providing witness languages defined over a binary alphabet. In the unary case, namely in the case of languages defined over a one-letter alphabet, it is an open question if these bounds remain valid. It is suitable to point out that in the unary case the classes of regular and context-free languages collapse [GR62] and, hence, d-limited automata are equivalent to finite automata for each $d > 0$. The existence of unary 1-limited automata which require a quadratic number of states to be simulated by two-way nondeterministic finite automata has been

proved [PP14]. The set of unary strings of length a multiple of 2^n can be recognized by a 2-limited automaton of size $O(n)$, for any fixed $n > 0$. On the other hand, each (even two-way nondeterministic) finite automaton requires a number of states exponential in n to accept the same language [PP15].

The investigation of the size of unary limited automata was deepened by Kutrib and Wendlandt: several bounds for the costs of the simulations of different variants of unary limited automata by different variants of finite automata were stated. Among these results, they proved the existence of unary languages accepted by $4n$-states deterministic 1-limited automata which require $n \cdot e^{\sqrt{n \ln n}}$ states to be accepted by two-way nondeterministic finite automata [KW15].

In this chapter we improve these results, by obtaining an exponential gap between unary deterministic 1-limited automata and two-way nondeterministic finite automata. To this aim, first we show that for each $n > 1$ the singleton language $\{ a^{2^n} \}$ can be recognized by a deterministic 1-limited automaton having $2n + 1$ states and a description of size $O(n)$. Since the same language requires $2^n + 1$ states to be accepted by a one-way nondeterministic automaton, it turns out that the state gap between deterministic 1-limited automata and one-way nondeterministic automata in the unary case is the same as in the binary case. Then, we shall also observe that the gap does not reduce if we want to convert unary deterministic 1-limited automata into two-way nondeterministic automata. However, when converting finite automata into limited automata, a size reduction corresponding to such a gap is not always achievable, even if we convert a unary one-way deterministic finite automaton into a nondeterministic d-limited automaton for any arbitrarily large d.

In the second part of the chapter, we consider unary context-free grammars. The cost of the conversion of these grammars into finite automata has been investigated by proving exponential gaps [PSW02]. Here, we study the conversion of unary context-free grammars into limited automata. With the help of a result presented by Okhotin [Okh12] (recalled in Theorem 2.2), we prove that each unary context-free grammar $\mathcal{G}$ can be con-

verted into an equivalent 1-limited automaton whose description has a size that is polynomial in the size of $\mathcal{G}$.

The results shown in this chapter have been presented in [PP19a].

5.1 On the Size of Unary Limited Automata

In this section we compare the sizes of unary limited automata with the sizes of equivalent finite automata. Our main result is that unary 1-LAs can be exponentially more succinct than finite automata even while comparing unary *deterministic* 1-LAs with *two-way nondeterministic* automata. However, there are unary regular languages that do not have any d-limited automaton which is significantly more succinct than finite automata, even for arbitrarily large d.

A large part of the section is devoted to showing that, for each $n \geq 0$, the language $U_n = \{ a^{2^n} \}$, which requires $2^n + 1$ states to be accepted by an NFA (even if it is able to scan the input in a two-way fashion), can be accepted by a deterministic 1-LA whose size is polynomial in n. Let us proceed by steps. In order to illustrate the construction, first it is useful to discuss how U_n can be accepted by a linear bounded automaton $\mathcal{M}_n$.[1]

$\mathcal{M}_n$ works in the following way:

i. Starting from the first input symbol, it scans the tape from left to right by counting modulo 2 the a's until the right end-marker is reached. Each odd-counted a is overwritten by X.

ii. The previous step is repeated $n - 1$ further times, after moving backward the head until reaching the left end-marker. If at the end of one of the iterations $\mathcal{M}_n$ discovers that the number of a's on the tape was odd then $\mathcal{M}_n$ rejects.

iii. After the last iteration, $\mathcal{M}_n$ accepts if only one a is left on the tape.

[1] We remind the reader that a linear bounded automaton is a Turing machine that can use as storage only the portion of the tape which initially contains the input, by rewriting its cells an unbounded number of times.

It is possible to modify $\mathcal{M}_n$, without any increasing of the number of the states, by introducing a different kind of writing at step i.: in the course of the i-th iteration, the symbol i is used instead of X. After the n-th iteration, only one cell of the tape should contain the symbol a. In this case, $\mathcal{M}_n$ writes the symbol n on such a cell and accepts; otherwise, $\mathcal{M}_n$ rejects. For example, in the case $n = 4$, at the end of computation the final contents of the tape on input a^{2^4} will be 0102010301020104. (Computations of $\mathcal{M}_n$ will be formally studied in Lemma 5.1.)

Considering the last extension of $\mathcal{M}_n$, we are now going to introduce a 1-LA $\mathcal{N}_n$ accepting the language U_n, based on the guessing of the final tape contents of $\mathcal{M}_n$.

In the first phase, $\mathcal{N}_n$ scans the tape replacing each a with a symbol nondeterministically chosen in $\{0, \ldots, n\}$. This requires only one state. Next, the machine, after moving backward the head to the left end-marker, makes a scan from left to right for each $i = 0, \ldots, n-1$, where it checks if the symbol i occurs in all odd positions, where positions are counted ignoring the cells containing numbers less than i. This control phase needs three states for each value of i: one for moving backward the head and two for counting modulo 2 the positions containing symbols greater or equal to i. Finally, the automaton checks if only the last cell contains n (two states), in such a case the input is accepted. The total number of states of $\mathcal{N}_n$ is $3(n+1)$, that, even in this case, is linear in the parameter n. This gives us a 1-LA of size polynomial in n accepting U_n.

We are now going to prove that we can do better. In fact, we will show that switching to the deterministic case for the limited automata model, the size of the resulting device does not increase. Actually, we will slightly reduce the number of states, while using the same working alphabet.

To this aim, we study the final tape contents of the linear bounded automaton $\mathcal{M}_n$ when it accepts the input:

Lemma 5.1. *At the end of the computation of $\mathcal{M}_n$ on input a^{2^n}, the j-th tape cell contains the exponent of the largest power of 2 that divides j, for $j = 1, \ldots, 2^n$.*

Proof. First, we prove by induction on $i = 0, \ldots, n-1$ that the cells that are not yet rewrit-

ten before iteration $i+1$ are those whose positions are multiples of 2^i. The basis, $i = 0$, is trivial. Furthermore, for $i = 0, \ldots, n-1$, the iteration $i+1$ rewrites the odd-counted cells among those which are not rewritten so far, namely, according to the induction hypothesis, among those in positions $1 \cdot 2^i, 2 \cdot 2^i, 3 \cdot 2^i, \ldots, 2^{n-i} \cdot 2^i$. Hence, the cells which are not still rewritten after iteration $i+1$ and before the next iteration (if any) are those whose positions are even multiples of 2^i, namely multiples of 2^{i+1}.

In this way, we can conclude that for $i = 0, \ldots, n-1$, iteration $i+1$ rewrites any cell of index j such that 2^i is the largest power of 2 that divides j. The proof can be completed just observing that the symbol used for rewriting in such iteration is i. □

From now on, let us denote by $\sigma_1\sigma_2 \cdots \sigma_j \cdots$ the infinite integer sequence which is defined by taking as j-th element σ_j the exponent of the highest power of 2 which divides j. This sequence is known as *binary carry sequence* [Sloa].[2]

From Lemma 5.1, it follows that the final tape contents of M_n (and of N_n), when the input is accepted, consists of the first 2^n elements of the binary carry sequence.

We notice that given integers $j > 0$, $k \geq 0$, $0 < j' < 2^k$, such that $j = 2^k + j'$, the exponents of the highest powers of 2 which divide j and j' are the same. Considering the definition of the sequence, this allows us to get the following equality, for all integers $j > 0, k \geq 0$:

$$\sigma_j = \begin{cases} k & \text{if } j = 2^k, \\ \sigma_{j-2^k} & \text{if } 2^k < j < 2^{k+1}. \end{cases} \tag{5.1}$$

Hence, the sequence can be iteratively obtained as follows:

- The first element of the sequence is 0.
- For $k \geq 0$, by making a copy of the first 2^k elements of the sequence and by replacing the last element in the copy by its successor, we obtain the next 2^k elements.

[2] In [AS03], the function associating with each integer j the exponent σ_j of the highest power of 2 which divides j is called the *ruler function*. A slightly different definition of the ruler function is given in [Slob].

For example, from 0, concatenating a copy and replacing the last (and unique) element of the copy by the successor, we obtain 01, from which, with the same process, we get 0102, 01020103, and so on.

Remark 5.1. *In the prefix of length 2^n of the binary carry sequence, each symbol i, $0 \leq i < n$, occurs 2^{n-i-1} times, starting in position 2^i and at distance 2^{i+1},* i.e., *it occurs in positions $2^i(2j-1)$, for $j = 1, \ldots, 2^{n-i-1}$. The symbol $i = n$ occurs in position 2^n only.*

Remark 5.1 is a direct consequence of the definition of the sequence. Consider, as an example, the sequence $x = 01020103$: reading x from left to right, the symbol 0 appears for the first time in position 2^0 and then in positions $3, 5, 7$; 1 occurs in positions $2, 6$; 2 occurs in position 4; and, finally, 3 occurs in position 8 only.

We will show that, for any $n > 0$, the prefix of length 2^n of this sequence can be generated by a deterministic 1-LA which writes it on its tape, *but avoids using large numbers*.

To this aim, we introduce the function `bis`, that associates with any given sequence of integers $s = k_1 k_2 \cdots k_j$, its *Backward Increasing Sequence*, namely the longest strictly increasing sequence which can be obtained by copying some elements from s, selected with the greedy strategy we now present. At the beginning the last element k_j of s is chosen as first element of `bis(`s`)`. Then the remaining elements are inspected from k_{j-1} to k_1, by appending one element to `bis(`s`)` only when it is greater than the last element added to `bis(`s`)`.

Formally, `bis(`$k_1 k_2 \cdots k_j$`)` $= (i_1, i_2, \ldots, i_r)$, $j, r > 0$, if and only if $i_1 = k_{h_1}, i_2 = k_{h_2}, \ldots, i_r = k_{h_r}$ where $h_1 = j$, $h_t = \max\{\, h' < h_{t-1} \mid k_{h'} > k_{h_{t-1}} \,\}$ for $t = 2, \ldots, r$, and $k_{h'} < k_{h_r}$ for $0 < h' < h_r$.

For example, considering the prefix $s = 01020103010$ of length $j = 11$ of the binary carry sequence, we obtain `bis(`s`)` by firstly selecting 0, namely the last element of s. Then, moving backwards, we select 1 (because $1 > 0$), we do not select 0 ($0 \leq 1$), we select 3 ($3 > 1$), and we do not select any of the remaining elements (all of them are not greater than 3). In this way, we finally get `bis(`01020103010`)` $= (0, 1, 3)$. Notice that in the binary representation of j, namely 1011, the bits set to 1 occur, respectively, in position 0, 1, and 3.

This fact is true for each j, as proved in the following lemma, *i.e.*, the value of `bis`, applied to the first j elements of the binary carry sequence, indicates the positions of bits equal to 1 in the binary representation of j, starting from the least significant bit.

We remind the reader that $\sigma_1\sigma_2\cdots\sigma_j$ denotes the prefix of length j of the binary carry sequence.

Lemma 5.2. *For $j > 0$, if $\texttt{bis}(\sigma_1\sigma_2\cdots\sigma_j) = (i_1, i_2, \ldots, i_r)$ then $j = \sum_{t=1}^{r} 2^{i_t}$.*

Proof. We proceed by induction on j.

- If j is a power of 2, namely $j = 2^k$, for some $k \geq 0$, then k is the maximum number in the sequence and, by definition, it occurs in position j only. So, $\texttt{bis}(\sigma_1\sigma_2\cdots\sigma_j) = (k)$.

- If j is not a power of 2, namely $2^k < j < 2^{k+1}$, $j = 2^k + j'$ for some $k > 0$, $0 < j' < 2^k$, then k is the maximum number which occurs in the sequence and, by equality (5.1), the subsequence $\sigma_{2^k+1}\sigma_{2^k+2}\cdots\sigma_j$ is equal to the subsequence $\sigma_1\sigma_2\cdots\sigma_{j'}$ of the first j' elements. Hence, $\texttt{bis}(\sigma_1\sigma_2\cdots\sigma_j)$ can be obtained by appending k at the end of $\texttt{bis}(\sigma_1\sigma_2\cdots\sigma_{j'})$, namely, if $\texttt{bis}(\sigma_1\sigma_2\cdots\sigma_{j'}) = (i_1, \ldots, i_{r'})$, $r' = r - 1$, then $\texttt{bis}(\sigma_1\sigma_2\cdots\sigma_j) = (i_1, \ldots, i_{r'}, i_r)$, $i_r = k$.

  By induction hypothesis, $j' = \sum_{t=1}^{r'} 2^{i_t}$. Thus $j = 2^k + j' = 2^k + \sum_{t=1}^{r'} 2^{i_t} = \sum_{t=1}^{r} 2^{i_t}$.

□

Using Lemma 5.2, we now prove a property which will be crucial in the construction of a deterministic 1-LA accepting the language U_n.

Lemma 5.3. *For $j > 0$, σ_j is the smallest integer greater than or equal to 0 not occurring in $\texttt{bis}(\sigma_1\sigma_2\cdots\sigma_{j-1})$.*

Proof. In the case $j = 1$, $\texttt{bis}(\sigma_1\sigma_2\cdots\sigma_{j-1})$ is empty and, hence, the statement of the lemma gives $\sigma_1 = 0$. To study the case $j > 1$, we first remind the reader that, by definition, σ_j is the exponent of the highest power of 2 which divides j, namely it coincides

with the position of the least significant 1 in the binary representation of j and, hence, with the lowest position which does not contain the digit 1 in the binary representation of $j-1$.[3] Considering Lemma 5.2, applied to $j-1$, we conclude that σ_j is the smallest integer not occurring in $\texttt{bis}(\sigma_1\sigma_2\cdots\sigma_{j-1})$. □

We are going to define a deterministic 1-LA $\mathcal{A}_n = \langle Q, \Sigma, \Gamma, \delta, q_I, F\rangle$ accepting the language $U_n = \{\, a^{2^n} \,\}$, for a fixed n. The automaton $\mathcal{A}_n$ writes on its tape the prefix of length 2^n of the binary carry sequence. In particular, in the first visit at the cell j, $\mathcal{A}_n$ writes the symbol σ_j of the binary carry sequence that, according to Lemma 5.3, can be computed as the smallest nonnegative integer missing in $\texttt{bis}(\sigma_1\sigma_2\cdots\sigma_{j-1})$. This computation is done by inspecting the part of the tape to the left of the j-th cell only. In this way, the contents of the cell j is changed in the first visit only.

The automaton $\mathcal{A}_n$ implements the procedure summarized in Procedure 8 — note that, for ease of presentation, the procedure assumes that the machine starts the computation with the head on the left end-marker — and it is defined as follows: $Q = \{\, q_I, q_F, q_1, \ldots, q_n, p_1, \ldots, p_{n-1} \,\}$, $\Sigma = \{\, a \,\}$, $\Gamma = \{\, \texttt{0}, \ldots, \texttt{n} \}$, q_I is the initial state and q_F is the unique final one. The transitions in δ are the following (undefined transitions are not listed):

i. $\delta(q_I, a) = (p_1, 0, -1)$

ii. $\delta(p_i, \sigma) = (p_i, \sigma, -1)$, for $i = 2, \ldots, n-1$ and $\sigma < i-1$

iii. $\delta(p_i, i) = (p_{i+1}, i, -1)$, for $i = 1, \ldots, n-2$

iv. $\delta(p_i, \sigma) = (q_i, \sigma, +1)$, for $i = 1, \ldots, n-1$ and ($\sigma > i$ or $\sigma = \triangleright$)

v. $\delta(p_{n-1}, n-1) = (q_n, n-1, +1)$

vi. $\delta(q_i, \sigma) = (q_i, \sigma, +1)$, for $i = 1, \ldots, n$ and $\sigma < i$

[3] Indeed the binary representation of j is $x10^k$, for some $k \geq 0$, $x \in \{0,1\}^*$, when the binary representation of $j-1$ is $x01^k$, or simply 1^k, if x is empty.

vii. $\delta(q_i, a) = (q_1, i, +1)$, for $i = 1, \ldots, n-1$

viii. $\delta(q_n, a) = (q_F, n, +1)$

ix. $\delta(q_F, \triangleleft) = (q_F, \triangleleft, +1)$

We finally observe that $\mathcal{A}_n$ has $2n+1$ states, which is linear in the parameter n.

Procedure 8: Recognition of the language U_n. Also in this case, the symbol σ denotes the symbol currently scanned by the head, which is automatically updated at each head move, while the special instruction REJECT causes the whole computation to halt and reject.

```
                                    /* start with the head on the left endmarker */
45 while σ ∉ {n, ◁} do
46     move the head to the right
47     write(0)
48     j ← 0
49     repeat
50         while σ ≤ j and σ ≠ ▷ do
51             move the head to the left
52         j ← j + 1
53     until σ ≠ j
54     repeat move the head to the right until σ = a
55     write(j)
56 if σ = n then
57     move the head to the right
58     if σ = ◁ then ACCEPT
59 REJECT
```

The machine starts in the initial state q_1. Since each symbol $\sigma \neq 0$ is preceded by 0 (a 0 occurs in each odd position), the automaton moves the head to the right and writes a 0

before each symbol in $\Gamma \setminus \{\,0\}$ (Transition i. — Lines 46 and 47). Every time the head is in a odd position p, the automaton has to look backward for the minimum integer j such that j is not in $\texttt{bis}(\sigma_1, \ldots, \sigma_p)$. This is done with Transitions from ii. to v. — Lines from 49 to 53. After that, $\mathcal{A}_n$ moves its head to the right until the first a is reached (Transitions vi. — Lines from 54 to 54) and writes the symbol j (Transitions vii. — Line 55). This is repeated until either the symbol n is written on the tape or the right end-marker is reached because the input length is less than 2^n. In the former case, it is sufficient to verify if the next symbol on the tape is the right end-marker: in this case, the automaton accepts (Transitions viii. and ix. — Lines from 56 to 58). In the latter case, $\mathcal{A}_n$ stops and rejects (undefined transition — Line 59).

Hence we conclude that the language U_n is accepted by a deterministic 1-LA with $O(n)$ states, while it is an easy observation that each 1NFA accepting it requires $2^n + 1$ states. We can even obtain a stronger result by proving that between unary deterministic 1-LAs and 2NFAs there is the same gap. Indeed, from Fact 5.2 in [Bir96], each 2NFA accepting U_n requires more than 2^n states. This gives the main result of this section:

Theorem 5.2. *For each integer $n > 1$ the language U_n is accepted by a deterministic* 1-LA *with $O(n)$ states and a working alphabet of size $O(n)$ while each* 2NFA *accepting it requires more than 2^n states.*

We conclude this section by proving that the exponential gap between unary limited automata and finite automata is not always achievable.

Theorem 5.3. *There exist constants c, n_0 such that for all integers $n \geq n_0$ there exists a unary* 1DFA *accepting a finite language L with at most n states, such that for any d-*LA *accepting L with $d > 0$, q states, and a working alphabet of m symbols, it holds that $qm \geq cn^{1/2}$.*

Proof. There are $2^{O(q^2m^2)}$ different limited automata such that the cardinalities of the set of states and of the working alphabet are bounded by q and m, respectively. On the other hand, the number of different subsets of $\{a^0, a^1, \ldots, a^{n-1}\}$ is 2^n. Hence $kq^2m^2 \geq n$ for a constant $k > 0$ and each sufficiently large n, which implies $qm \geq cn^{1/2}$, where $c = 1/k^{1/2}$.

Notice that each subset of $\{ a^0, a^1, \ldots, a^{n-1} \}$ is accepted by a (possibly incomplete) 1DFA with at most n states. □

The result in Theorem 5.3 does not depend on d, *i.e.*, the lower bound holds even taking an arbitrarily large d. In the case $d = 1$, the argument in the proof can be refined to show that $qm^{1/2} \geq cn^{1/2}$.

5.2 Unary Grammars versus Limited Automata

In Section 5.1 we proved an exponential gap between unary 1-LAs and finite automata. A similar gap was obtained between unary CFGs and finite automata [PSW02]. Hence, it is natural to study the size relationships between unary CFGs and 1-LAs. Here, we prove that each context-free grammar $\mathcal{G}$ specifying a unary language can be converted into an equivalent 1-LA $\mathcal{M}$ of polynomial size. More precisely, the sizes of the set of states and of the working alphabet of $\mathcal{M}$ are polynomial with respect to the size of $\mathcal{G}$.

Let us start by presenting some notions and preliminary results.

Definition 5.1. *Given an integer $d > 0$, the* extended Dyck language with nesting depth bounded by *d over Ω, denoted as $\widehat{\mathcal{D}}_{\Omega}^{(d)}$, is the subset of $\widehat{\mathcal{D}}_{\Omega}$ consisting of all strings where the nesting depth of brackets is at most d.*

Example 5.1. Let $\Omega_k = \{ (, [,),] \}$, $\Omega_n = \{ | \}$, and $\Omega = \Omega_k \cup \Omega_n$. Then $|([[]]|)[|] \in \widehat{D}_{\Omega}^{(3)} \setminus \widehat{D}_{\Omega}^{(2)}$. ■

As discussed in Section 2.2.2.1, it is well-known that Dyck languages, and so extended Dyck languages, are context free and nonregular. However, the subsets obtained by bounding the nesting depth by any fixed constant are regular. We are interested in the recognition of such languages by "small" two-way automata:

Lemma 5.4. *Given an extended bracket alphabet Ω with k types of brackets and an integer $d > 0$, the language $\widehat{\mathcal{D}}_{\Omega}^{(d)}$ can be recognized by a* 2DFA *with $O(k \cdot d)$ states.*

Proof. We can define a 2DFA $\mathcal{M}$ which verifies the membership of its input w to $\widehat{\mathcal{D}}_{\Omega}^{(d)}$ by using a counter c. During a first scan $\mathcal{M}$ checks whether or not the brackets are correctly nested, *regardless* their types. This is done as follows. Starting with 0 in c, $\mathcal{M}$ scans the input from left to right, incrementing the counter for each left bracket and decrementing it for each right bracket. If during this process the counter exceeds d or becomes negative then $\mathcal{M}$ rejects. $\mathcal{M}$ also rejects if at the end of this scan the value stored in the counter is positive.

In the remaining part of the computation, $\mathcal{M}$ verifies that corresponding left and right brackets are of the same type. To this aim, starting from the left end-marker, $\mathcal{M}$ moves its head to the right, to locate a left bracket. When it is found, $\mathcal{M}$ saves it in the finite control and moves to the right to locate the corresponding right bracket. This is done by using the counter c, which is set to 0 on the left bracket and it is incremented or decremented for each left or right bracket, respectively, which is encountered while moving to the right. In this way, the right bracket which is reached when c contains 0 corresponds to the left bracket under inspection. When such right bracket is reached, $\mathcal{M}$ verifies the matching with the one saved in the control. If this is not the case, then $\mathcal{M}$ stops and rejects. Otherwise, $\mathcal{M}$ should move back its head to the matched left bracket in order to continue the inspection. This can be done, using the same method, by moving the head to the left and incrementing or decrementing the counter for each right or left bracket, respectively, up to reach a cell containing a left bracket when 0 is in c. At this point, $\mathcal{M}$ moves to the right to locate the next left bracket and to check the matching with its corresponding right bracket by the same procedure.

This process is repeated up to reach the right end-marker. At that point, all pairs of brackets have been inspected. Notice that neutral symbols are completely ignored.

In its finite control, $\mathcal{M}$ keeps the counter c, that can assume $d+1$ different values, and can store a left bracket. This yields $O(k \cdot d)$ states. □

It is possible to notice that the computation of the obtained 2DFA $\mathcal{M}$ accepting $\widehat{\mathcal{D}}_{\Omega}^{(d)}$ can be made sweeping (namely, in which the head can change direction only on the end-

markers), even rotating (a sweeping computation in which the head is reset to the left endmarker every time the right endmarker is reached). After the first scan (or *sweep*) during which $\mathcal{M}$ checks whether or not the brackets are correctly nested, regardless their types, h further sweeps are performed. In particular, during the i-th sweep, for $i = 1, \ldots, d$, $\mathcal{M}$ checks the types of the brackets at level i, by using a counter kept in the finite state control to keep track of the current nesting level of the brackets. This increases the number of the states to $O(k \cdot d^2)$.

Corollary 5.1. *Given an extended bracket alphabet Ω with k types of brackets and an integer $d > 0$, the language $\widehat{\mathcal{D}}_{\Omega}^{(d)}$ can be recognized by a sweeping (rotating)* 2DFA *with $O(k \cdot d^2)$ states.*

In the following we shall use the nonerasing variant of the Chomsky-Schützenberger representation theorem for context-free languages, proved by Okhotin (*cf.*, Theorem 2.2) to obtain our main result.

Pighizzini observed that the size of the alphabet Ω is polynomial with respect to the size of a context-free grammar $\mathcal{G}$ generating L and the language R of Theorem 2.2 is local [Pig16], namely, such that there exist sets $\mathcal{A} \subseteq \Sigma \times \Sigma$, $\mathcal{I} \subseteq \Sigma$, and $\mathcal{F} \subseteq \Sigma$ such that $w \in R$ if and only if all factors of length 2 in w belong to $\mathcal{A}$ and the first and the last symbols of w belong to $\mathcal{I}$ and $\mathcal{F}$, respectively [MP71].[4]

This was used to prove that each context-free grammar $\mathcal{G}$ can be transformed into an equivalent *strongly limited automaton* (a special kind of 2-LA) whose description has polynomial size with respect to the description of $\mathcal{G}$. In the following, when L is specified by a context-free grammar $\mathcal{G}$, *i.e.*, $L = L(\mathcal{G})$, we will write $\Omega_{\mathcal{G}}$ and $R_{\mathcal{G}}$ instead of Ω_L and R_L, respectively.

Our goal, here, is to build 1-LAs of polynomial size from unary context-free grammars. To this aim, using the fact that factors in unary strings commute, by adapting the argument used to obtain Theorem 2.2, we prove the following result:

[4]It is not difficult to verify that local languages are a subclass of regular languages.

Theorem 5.4. *Let $L \subseteq \{\, a \,\}^*$ be a unary regular language and $G = \langle V, \{\, a \,\}, P, S \rangle$ be a context-free grammar of size s generating it. Then, there exist an extended bracket alphabet $\Omega_{\mathcal{G}}$ and a regular language $\widehat{R}_{\mathcal{G}} \subseteq \Omega_{\mathcal{G}}^*$ such that $L = h(\widehat{\mathcal{D}}_{\Omega_{\mathcal{G}}}^{(\#V)} \cap \widehat{R}_{\mathcal{G}})$, where:*

- *$\widehat{\mathcal{D}}_{\Omega_{\mathcal{G}}}^{(\#V)}$ is the extended Dyck language over $\Omega_{\mathcal{G}}$ with nesting depth bounded by $\#V$,*
- *h is the letter-to-letter homomorphism which maps each element of $\Omega_{\mathcal{G}}$ into the symbol a.*

Furthermore, the size of $\Omega_{\mathcal{G}}$ is polynomial in the size s of the grammar $\mathcal{G}$ and the language $\widehat{R}_{\mathcal{G}}$ is recognized by a 2DFA *with a number of states polynomial in s.*

Proof. Here we present an outline of the argument used to prove the result. A detailed proof of a more general result is given in [PP19a].

Given a context-free grammar $\mathcal{G} = \langle V, \{\, a \,\}, P, S \rangle$ specifying a unary language L, we first obtain the representation in Theorem 2.2. According to Theorem 5.2 in [Pig16], the size of the alphabet $\Omega_{\mathcal{G}}$ is polynomial with respect to the size of the description of $\mathcal{G}$. Each pair of brackets in $\Omega_{\mathcal{G}}$ represents the root of a derivation tree of $\mathcal{G}$, which starts from a certain variable of $\mathcal{G}$ and produces a terminal string.

If a sequence $w \in \Omega_{\mathcal{G}}^*$ contains a pair of brackets corresponding to a variable A which is nested, at some level, in another pair corresponding to the same variable, then w can be replaced by a sequence w' of the same length, which is obtained by replacing the factor of w delimited by the outer pair of brackets corresponding to A, by the factor delimited by the inner pair, and by moving the removed part at the end of w. For instance, consider the sequence $w = (_S\,(_A\,(_B\,)_B\,(_C\,(_A\,(_B\,)_B\,)_A\,)_C\,)_A\,)_S$ where, for the sake of simplicity, subscripts represent variables corresponding to brackets. The factor delimited by the pair $(_A\,)_A$ at the inner level is $(_A\,(_B\,)_B\,)_A$ which can replace the factor $(_A\,(_B\,)_B\,(_C\,(_A\,(_B\,)_B\,)_A\,)_C\,)_A$, which is delimited by the same pair at the outer level. Moving the remaining part $(_A\,(_B\,)_B\,(_C\,)_C\,)_A$ at the end, we obtain $w' = (_S\,(_A\,(_B\,)_B\,)_A\,)_S\,(_A\,(_B\,)_B\,(_C\,)_C\,)_A$. In such a way, each time the nesting depth is greater than $\#V$, it can be reduced by repeatedly moving some part to the end. So, from each string in $\widehat{\mathcal{D}}_{\Omega_{\mathcal{G}}}$, we can obtain an "equivalent" string of the same length in $\widehat{\mathcal{D}}_{\Omega_{\mathcal{G}}}^{(\#V)}$.

The regular language $R_{\mathcal{G}}$ should be modified accordingly. While in the representation in Theorem 2.2, the first and the last symbol of a string $w \in \widehat{\mathcal{D}}_{\Omega_{\mathcal{G}}} \cap R_{\mathcal{G}}$ represent a matching pair corresponding to the variable S, after the above transformation, valid strings should correspond to sequences of blocks of brackets where the first block represents a derivation tree of a terminal string from S, while each of the subsequent blocks represents a *gap tree* from a variable A, namely a tree corresponding to a derivation of the form $A \stackrel{+}{\Rightarrow} a^i A a^j$, with $i + j > 0$, where A already appeared in some of the previous blocks. This condition, together with the conditions on $R_{\mathcal{G}}$, can be verified by a 2DFA with a polynomial number of states. □

Notice that if we omit the state bound for the 2DFA accepting $\widehat{R}_{\mathcal{G}}$, the statement of Theorem 5.4 becomes trivial: by taking $\Omega_{\mathcal{G}} = \{ a \}$ where a is a neutral symbol, $R_{\mathcal{G}} = L$, and $h(a) = a$, we obtain $\widehat{\mathcal{D}}^{(\#V)}_{\Omega_{\mathcal{G}}} = \{ a \}^*$ and hence $\widehat{\mathcal{D}}^{(\#V)}_{\Omega_{\mathcal{G}}} \cap \widehat{R}_{\mathcal{G}} = L = h(L)$.

Using Theorem 5.4, we now prove the main result of this section:

Theorem 5.5. *Each context-free grammar of size s generating a unary language can be converted into an equivalent* 1-LA *having a size that is polynomial in s.*

Proof. Let $\mathcal{G} = \langle V, \{ a \}, P, S \rangle$ be the given grammar, $L \subseteq \{ a \}^*$ be the unary language generated by it, $\Omega_{\mathcal{G}}$ be the extended bracket alphabet, and $\widehat{R}_{\mathcal{G}}$ be the regular language obtained from $\mathcal{G}$ according to Theorem 5.4.

We define a 1-LA $\mathcal{M}$ which works as follows:

1. $\mathcal{M}$ makes a complete scan of the input tape from left to right, by rewriting each input cell by a nondeterministically chosen symbol from $\Omega_{\mathcal{G}}$. Let $w \in \Omega^*_{\mathcal{G}}$ be the string written on the tape at the end of this phase.
2. $\mathcal{M}$ checks whether or not $w \in \widehat{\mathcal{D}}^{(\#V)}_{\Omega_{\mathcal{G}}}$.
3. $\mathcal{M}$ checks whether or not $w \in \widehat{R}_{\mathcal{G}}$.
4. $\mathcal{M}$ accepts if and only if the outcomes of steps 2 and 3 are both positive.

According to Lemma 5.4, Step 2 can be done by simulating a 2DFA with $O(\#\Omega_{\mathcal{G}} \cdot \#V)$ states, hence a number polynomial in s. Furthermore, by Theorem 5.4, also Step 3 can be performed by simulating a 2DFA with a number of states polynomial in s. Hence $\mathcal{M}$ has a size that is polynomial in s. □

We point out that from Theorem 5.5 and the exponential gap from unary CFGs to 1NFAs proved in [PSW02], we could derive an exponential gap from unary *nondeterministic* 1-LAs to 1NFAs. In Section 5.1 we proved that the gap remains exponential if we restrict to unary *deterministic* 1-LAs and consider the simulation by 2NFAs.

5.3 Remarks

Using languages defined over a binary alphabet, exponential size gaps were proved for the conversion of 1-LAs into 2NFAs and of deterministic 1-LAs into 1DFAs [PP14]. As a consequence of our results, these exponential gaps hold even in the restricted case of unary languages. On the other hand, the gap between sizes of 1-LAs and 1DFAs is doubly exponential. Even in this case, the proof given by Pighizzini and Pisoni relies on witness languages defined over a binary alphabet. We leave as an open question to investigate whether or not a double exponential gap is possible between 1-LAs and 1DFAs even in the unary case.

Another question we leave open is whether or not 1-LAs and CFGs are polynomially related in the unary case. While in Section 5.2 we proved that from each unary CFG we can build a 1-LA of polynomial size, at the moment we do not know the converse relationship. The same question can be formulated by dropping the restriction to the unary case. We point out that, in the general case, the size cost of the conversion of 2-LAs into equivalent CFGs is exponential [PP15]. The cost remains exponential when we convert d-LAs into CFGs, for each $d > 2$ [KPW18].

Furthermore, it would be interesting to know the costs of the conversions when deterministic devices are considered. For instance, the 1-LAs produced by our conversion

from unary CFGs strongly rely on the use of nondeterministic choices. So, it would be interesting to know what is the size cost if we want to obtain *deterministic* 1-LAs.

6 Limited Automata: A Time Constraint

As observed by Hennie in 1965, deterministic one-tape Turing machines operating in linear time recognize exactly the class of regular languages [Hen65]. The result has later been extended to the nondeterministic case [TYL10, Pig09b]. Here, operating in linear time means that every computation has length linearly bounded in the input length. In particular, linear-time machines are necessarily halting — see [Pig09b] for investigations of alternative linear time restrictions. The above-mentioned result implies that every Hennie machine is equivalent to some finite automaton. From the opposite point of view, this means that providing two-way finite automata with the ability to overwrite the tape cells does not extend the expressiveness of the model, as long as the time is linearly bounded in the length of the input.

However, Průša showed that it is undecidable given a bounded deterministic Turing machine to check whether it works in linear time over all input strings, namely, whether it is actually a deterministic Hennie machine [Prů14]. To avoid this drawback, he proposed the weight-reducing variant of Hennie machines, in which the time limitation is syntactic (refer to Section 2.2 for the definitions of these models). As a consequence, the number of visits of a cell by the head is bounded by some constant (*i.e.*, not depending on the input length) hence the device works in linear time over every input string.

By contrast to Hennie machines, in d-limited automata the head is allowed to visit a cell after the d-th visit, even if it cannot rewrite the contents anymore. This allows to

use super-linear time. Hence, limited automata live midway between linear-bounded automata and weight-reducing Hennie machines.

Also in the case $d = 1$, 1-limited automata can operate in super-linear time (as we shall show in Example 6.1). This contrasts with Hennie machines which operate in linear time by definition. The question we address in this chapter is whether this ability of 1-limited automata with respect to Hennie machines yields a gap between the two models in terms of the size of their representations.

We show that, with a polynomial increase in size, each 1-limited automaton can be transformed into an equivalent linear-time 1-limited automaton, or, alternatively, into a weight-reducing Hennie machine. Furthermore, we are able to obtain a deterministic device when the original machine is deterministic as well. We also show that the 1-limited automata resulting from our constructions have a special structure that can be exploited in order to obtain equivalent 1-limited automata in which an initial phase just overwrites each tape cell, *i.e.*, the device initially performs a nondeterministic left-to-right pass over the tape during which all the cells are independently overwritten. Similar behaviors have been considered in the context of *regular transduction*, because of their correspondence with global existential quantification in *monadic second order logic*, see [Boj+17] in which the authors define an operation called *common guess* corresponding to a nondeterministic sweep from left to right, overwriting the tape. Hence, as a consequence of our main result, each 1-limited automaton can be simulated by a *two-way automaton with common guess* of polynomial size. It follows that *reversing* a 1-limited automaton, *i.e.*, transforming it into another one recognizing the reverse of its accepted language, has polynomial cost only. This fails in the deterministic case, for which we exhibit an exponential lower bound. As a consequence, we obtain exponential lower bounds for the simulation of deterministic weight-reducing Hennie machines by deterministic 1-limited automata. The results are summarized in Figure 6.1.

The chapter is organized as follows. In Section 6.1 we start by presenting a simpler form of 1-LAs and a preliminary example showing that 1-LAs can operate in super-linear

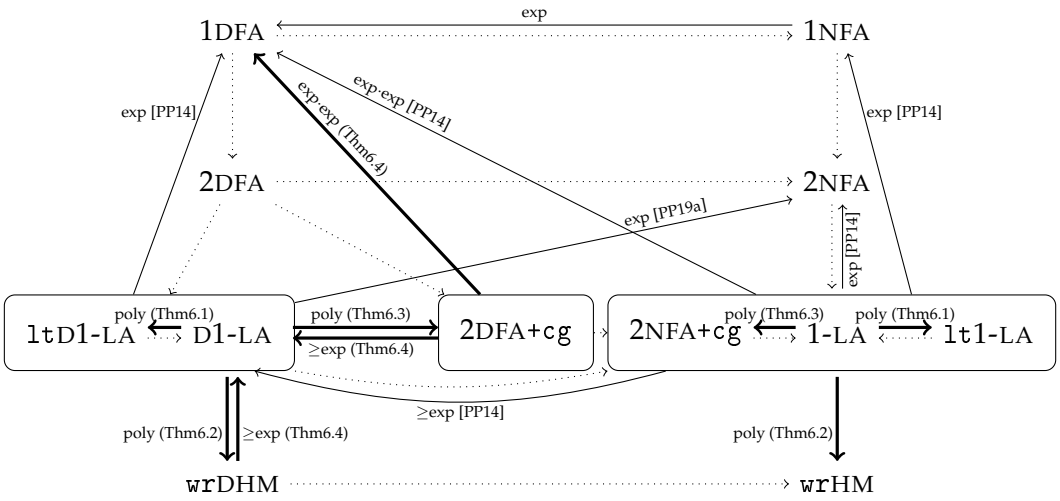

Figure 6.1: Relationships between the main models studied in the chapter. Here, `lt` and `wr` mean linear-time and weight-reducing, while D1-LA and (D)HM stand for deterministic 1-LA and (deterministic) Hennie machine, respectively. Deterministic and nondeterministic two-way automata with common guess are denoted by 2DFA+cg and 2NFA+cg. Dotted arrows indicate trivial connections while thick arrows indicate our results.

time. The main constructions used for proving our results are detailed in Section 6.2, while the results are finally presented in Section 6.3.

The results discussed in this chapter have been presented in [GP19].

6.1 Preliminaries

The following result, which is instrumental for our later proofs, gives a simpler form of 1-LAs.

Lemma 6.1. *For each n-state* 1-LA*, there exists an equivalent* $3n$*-state* 1-LA *using the same working alphabet, which performs stationary moves exactly when rewriting a cell content. Furthermore, the conversion preserves determinism.*

Proof. First, by using standard techniques, each sequence of stationary moves followed by a nonstationary move can be replaced by a nonstationary transition. This operation

does not increase the size of the 1-LA. Since in every accepting computation the last transition performed by any 1-LA is a right move from the rightmost cell of the tape, this modification eliminates all the stationary moves.

Second, we can split every rewriting step into two steps: a first stationary step during which the input cell is rewritten, followed by a second read-only nonstationary step. This yields a linear increase of the size of the device only. More precisely, for each state $q \in Q$ and each direction $d \in \{ -1, +1 \}$, we create a new copy $(q, d) \notin Q$ of q, whose interpretation is to delay the head move d. Then, each rewriting transition $(q, \gamma, d) \in \delta(p, \sigma)$ is replaced by two transitions $((q, d), \gamma, 0)$ from p on σ and (q, γ, d) from (q, d) on γ. □

Example 6.1. Let us consider the family of languages $(L_n)_{n=0}^{\infty}$, in which, for each $n \in \mathbb{N}$,

$$L_n = \{ x_0 x_1 \cdots x_k \mid k \in \mathbb{N}, \text{ for each } i\colon\ x_i \in \{ a, b \}^n, \text{ for some } j \neq 0\colon\ x_j = x_0 \}.$$

A deterministic 1-LA $\mathcal{A}_n$ may recognize L_n as follows. It first scans the factor x_0, overwriting each input symbol with a marked copy. Then, $\mathcal{A}_n$ repeats a subroutine which overwrites a factor x_i with some fixed symbol $\sharp$, while checking in the meantime whether x_i equals x_0 or not. This can be achieved as follows. Before overwriting the j-th symbol of x_i, first, $\mathcal{A}_n$, with the help of a counter modulo n, moves the head leftward to the position j of x_0 and stores the unmarked scanned symbol σ in its finite control; second, it moves the head rightward until reaching the position j of x_i, namely, the leftmost position that has not been overwritten so far. At this point, $\mathcal{A}_n$ compares the scanned symbol (*i.e.*, the j-th symbol of x_i) with σ (*i.e.*, the j-th symbol of x_0). By setting a Boolean variable to **true** when complete factor x_i has matched x_0 and finally checking that the input string has length multiple of n, $\mathcal{A}_n$ can decide the membership of the input to L_n.

It is possible to implement $\mathcal{A}_n$ with a number of states linear in n and $\#\Sigma + 1$ working symbols. Since for each position of a factor x_i, $i > 0$, the head has to move back to the factor x_0, we observe that $\mathcal{A}_n$ works in quadratic time in the length of the input string. ■

6.2 Linear-Time Simulations of 1-Limited Automata

If a linear-space Turing machine can visit a tape cell only a constant number of times, it necessarily works in linear time. Conversely, Turing machines working in linear time (*i.e.*, Hennie machines), have been shown to visit each tape cell only a constant number of times during a computation [Hen65]. This contrasts with the case of 1-LAs, which can use quadratic time, as shown in Example 6.1. However, our main contribution states that, with a polynomial increase in size of the model, we can recover the above property, and therefore obtain equivalent 1-LAs working in linear time.

6.2.1 Main ingredients

6.2.1.1 Local window space bound

The key idea to obtain a linear-time bound, is to ensure that, in any computation, the simulating device works on a virtual window of fixed size that is shifted along the tape in a one-way fashion. More precisely, in the computations of our simulating 1-LAs, there exists a constant K not depending on the input length, such that, for any two tape cells at distance K, the leftmost one cannot be visited after having visited the rightmost one. In this way, it is possible to bound the number of visits of each cell.

In our simulation we divide the input word into blocks of some fixed length ℓ, given by some polynomial in the number n of states of the simulated 1-LA. Then, our virtual window covers two successive blocks, *i.e.*, $K = 2\ell$. The length ℓ is chosen in such a way that, once overwritten, a block on the tape may contain the sufficient information for recovering the behaviors of the simulated machine that may occur on the left of the window. Describing and storing this information is the purpose of the following subsection.

6.2.1.2 Shepherdson tables

Pighizzini and Pisoni presented a construction to simulate any 1-LA $\mathcal{A}$ by a finite automaton $\mathcal{B}$ [PP14], using classic ideas from the simulation of two-way automata by one-way

automata [She59]. The main ingredient was to store in the finite control of $\mathcal{B}$, a "transition table" describing the possible behaviors of $\mathcal{A}$ that may occur to the left of the current head position. Since the part of the tape to the left of the current head position has necessarily already been visited, its "frozen" content belongs to $\rhd(\Gamma \setminus \Sigma)^* \cup \{ \varepsilon \}$. Hence, the above-mentioned behaviors to the left of the current head position are read-only computations. To represent them, for each word $z'X \in \rhd(\Gamma \setminus \Sigma)^*$ with $|X| = 1$, we consider a relation $\tau_{z'X} \subseteq Q \times Q$, where Q denotes the set of states of $\mathcal{A}$. A pair (p, q) belongs to $\tau_{z'X}$ if and only if, starting from state p with the head scanning the last symbol of $z'X$, $\mathcal{A}$ may reach state q one cell to the right of $z'X$. Formally,

$$\tau_{z'X} = \{ (p, q) \mid z'pX \vdash^* z'Xq \},$$

where $z'pX$ and $z'Xq$ are partial configurations of $\mathcal{A}$ (see Section 2.2.1.2), and $z'pX \vdash^* z'Xq$ means that there exists a computation path

- starting from the rightmost position of $z'X$ (labeled by X) in state p,
- ending one cell to the right of this position in state q, and
- which visits only cells from the part of the tape containing $z'X$ in the meantime.

With the information of $\tau_{z'X}$, $\mathcal{B}$ has no need to read the part of the tape containing $z'X$, that is, to move its head leftward. Furthermore, given a symbol σ and a string $z = z'X$, we can construct $\tau_{z\sigma}$ from τ_z by scanning σ.[1] This is achieved by observing that (see Figure 6.2) $(p, q) \in \tau_{z\sigma}$ if and only if there exists a sequence $r_0, s_0, r_1, s_1, \ldots, r_\ell \in Q$, with $\ell \geq 0$, satisfying:

- $r_0 = p$,
- $(q, +1) \in \delta(r_\ell, \sigma)$, and
- $(s_i, -1) \in \delta(r_i, \sigma)$ and $(s_i, r_{i+1}) \in \tau_z$, for $i = 0, \ldots, \ell - 1$.

[1] For convenience, we set $\tau_\varepsilon = \emptyset$.

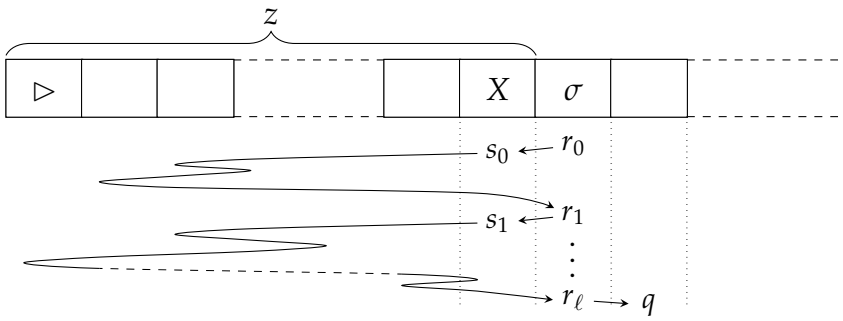

Figure 6.2: A computation path from $p = r_0$ to q giving $(p, q) \in \tau_{z\sigma}$. For each i, $(s_i, r_{i+1}) \in \tau_z$ and $(s_i, -1) \in \delta(r_i, \sigma)$, while $(q, +1) \in \delta(r_\ell, \sigma)$.

We denote by n the cardinality of Q. In the simulation of 1-LAs by finite automata, the table of size n^2 corresponding to the relation $\tau_{z'X}$ was stored in the finite control of the simulating 1NFA $\mathcal{B}$ and it was updated at each step. This yielded an exponential number of states for storing the 2^{n^2} possible tables thus implying an exponential size of $\mathcal{B}$ with respect to $\mathcal{A}$. This blowup was shown to be necessary in the worst case for the considered simulation [PP14].

Here, as our simulating device is a 1-LA, we take advantage of its ability to write on the tape. We indeed store the table $\tau_{z'X}$ onto the n^2 cells following the last position of $z'X$. Formally, fixing a bijection μ from $\{0, \ldots, n^2 - 1\}$ to $Q \times Q$, the i-th cell of the portion storing the table $\tau_{z'X}$ will contain 1 if $\mu(i)$ belongs to $\tau_{z'X}$, and 0 otherwise. As a consequence, we do not store all the tables corresponding to each tape position but a subcollection of them. More precisely, we only store tables $\tau_{z'X}$ for tape content prefixes $z'X$ of length multiple of n^2. Thus, updating the tables should be done block by block rather than cell by cell, for a decomposition of the input into blocks of length n^2. (We consider the cell containing the left endmarker as a complete block, while the last block containing the right endmarker may be shorter than n^2.)

When $\mathcal{A}$ is deterministic, $\tau_{z'X}$ is a partial[2] function from Q to Q. In this case it is

[2]In the deterministic case, the image associated with p by $\tau_{z'X}$ is undefined if one of the two following cases of the computation starting in $z'pX$ occurs: either, after a finite number of steps, no successive

possible to improve the above-described construction by storing the tables $\tau_{z'X}$ on the n cells following the last position of $z'X$. The input is therefore decomposed into blocks of length n rather than n^2. However, the alphabet used to store the table has size $n+1$ rather than 2. Indeed, the i-th cell of the portion storing the table will contain the image of the i-th state of $\mathcal{A}$ by $\tau_{z'X}$ if defined, or a special symbol $\perp \notin Q$ otherwise.

6.2.2 Construction of the simulating 1-LA $\mathcal{A}'$

Our simulation combines the two ideas discussed previously, by storing a subcollection of the Shepherdson tables on the tape. We actually present several simulations transforming 1-LAs into equivalent linear-time 1-LAs. The most general one produces a nondeterministic 1-LA from a nondeterministic 1-LA. The other ones produce a deterministic 1-LA from a deterministic 1-LA. The various constructions are very similar and differ only in some basic routines and in the encoding of the Shepherdson tables. We first present their common global structure and then we specify their differences, when detailing the low-level implementation of the basic operations and subroutines used for the simulation in Section 6.2.2.3. To this end, we now fix some convenient notations.

Let $\mathcal{A} = \langle Q, \Sigma, \Gamma, \delta, q_0, F \rangle$ be the source 1-LA. By Lemma 6.1, modulo a linear size increase, we suppose that $\mathcal{A}$ performs stationary moves exactly when overwriting a cell content. Our goal is to build a linear-time 1-LA $\mathcal{A}' = \langle Q', \Sigma, \Gamma', \delta', q'_0, F' \rangle$ equivalent to $\mathcal{A}$, which has polynomial size with respect to to the size of $\mathcal{A}$.

Let ℓ denote the size of the blocks in the tape decomposition discussed above, and let T denote the set of symbols used to encode the Shepherdson tables on tape. Formally, either $\ell = n^2$ and $T = \{0,1\}$, or, possibly if $\mathcal{A}$ is deterministic, $\ell = n$ and $T = Q_\perp$ where $Q_\perp = Q \cup \{\perp\}$ for $\perp$ a symbol not belonging to Q. Moreover, we fix a mapping ν from $\{0,\ldots,\ell-1\}$ to Q defined for each index $i \in \{0,\ldots,\ell-1\}$ as follows:

- if $\ell = n^2$ then $\nu(i)$ is the state p such that $\mu(i) = (p,q)$ for some state q (recall that μ

transition is defined (incompleteness of $\mathcal{A}$), or the computation eventually enters a deterministic loop (non-haltingness of $\mathcal{A}$).

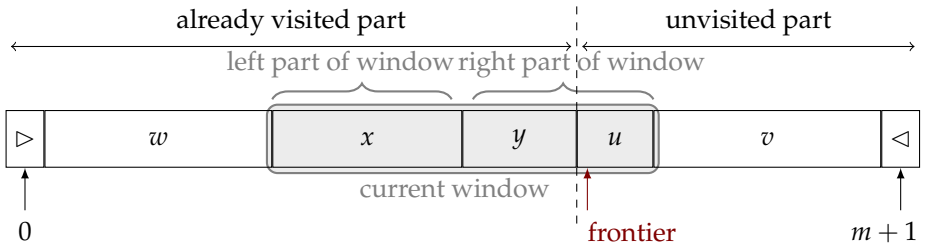

Figure 6.3: Typical description of the window during a computation of $\mathcal{A}$: m denotes the length of the input word, the current frontier occurs in the right block as first position of u, $w \in ((\Gamma \setminus \Sigma)^\ell)^*$, $x \in (\Gamma \setminus \Sigma)^\ell$, $y \in (\Gamma \setminus \Sigma)^*$, $u \in \Sigma^+$ with $|yu| = \ell$, and $v \in \Sigma^*$.

is a fixed bijection from $\{ 0, \ldots, n^2 - 1 \}$ to Q^2);

- if $\ell = n$ then $\nu(i) = q_i$, assuming $Q = \{ q_0, \ldots, q_{n-1} \}$.

During a computation of our simulating 1-LA $\mathcal{A}'$, the frozen content of the tape can be viewed as divided into two tracks: the first track contains the symbols overwritten by $\mathcal{A}$ in the simulated computation; the second track contains the encoding of the Shepherdson tables $\tau_{z'X}$. We thus fix the set of working symbols to be the product of $\Gamma \setminus \Sigma$ and T, *i.e.*,

$$\Gamma' = ((\Gamma \setminus \Sigma) \times T) \cup \Sigma.$$

As previously explained, the behavior of $\mathcal{A}'$ will be locally restricted to a window of bounded width. At any time in a computation of $\mathcal{A}$ we consider a virtual window which covers two successive blocks in the tape decomposition described above. The right block covered by the window contains the leftmost cell that has not been visited so far, to which we refer as current *frontier*. The content x of the left block covered by the window belongs to $(\Gamma \setminus \Sigma)^\ell \cup \{ \rhd \}$. The content of the right block covered by the window can be decomposed into yu with $y \in (\Gamma \setminus \Sigma)^*$ and $u \in \Sigma^+ \cup \Sigma^* \lhd$ such that $|yu| = \ell$ unless, possibly, $\lhd$ occurs in u, in which case $|yu| \leq \ell$. The frontier is on the first position of u. A typical situation is depicted in Figure 6.3.

In order to simulate $\mathcal{A}$, the linear-time 1-LA $\mathcal{A}'$ overwrites each block with a word $\widetilde{x} \in (\Gamma' \setminus \Sigma)^\ell$ whose projection on $\Gamma \setminus \Sigma$ is the word x written by $\mathcal{A}$ on the corresponding

block in the simulated computation, and the projection on T is exactly the encoding of the table τ_z, where z is the content of the tape to the left of the corresponding block in the simulated computation. (In Figure 6.3, $z = \triangleright w$ when considering the left block covered by the window, whose frozen content is x.) Roughly, in the simulating computation, when the frontier occurs at the first position of a block, $\mathcal{A}'$ has to fill this block, cell by cell, with the encoding of τ_{zx}, where x (resp. z) is the projection on $\Gamma \setminus \Sigma$ of the content $\widetilde{x}$ of the preceding block which is covered by the window (resp. of the content $\widetilde{z}$ of the tape to the left of the current window). To this end, it has read-only access to the left block, whose content $\widetilde{x}$ gathers all the required information, namely τ_z and x. In parallel, $\mathcal{A}'$ should also recover the simulated computation of $\mathcal{A}$. As soon as the right block is completely filled, the window is shifted to the right, in such a way that it covers the block just treated (as left part) and its successor (as right part).

6.2.2.1 Auxiliary procedures `readFromTable` and `simulateLeft`

Our simulation uses two subroutines, `readFromTable` and `simulateLeft`, which are called in order to recover some value from τ_z, where z is a tape content prefix in the simulated computation. Using the above-given notations (see, *e.g.*, Figure 6.3), we suppose that the virtual window covers two successive blocks, the left one containing $\widetilde{x} \in (\Gamma' \setminus \Sigma)^{\ell} \cup \{ \triangleright \}$, and the right one being partially filled with a prefix $\widetilde{y} \in (\Gamma' \setminus \Sigma)^{*}$ of length less than ℓ. We denote by $z \in \triangleright((\Gamma \setminus \Sigma)^{\ell})^{*} \cup \{ \varepsilon \}$ the tape content to the left of the window in the simulated computation.

`readFromTable` starts from and ends in the first position of the block containing $\widetilde{x}$ with a state p given as argument and returns a state q such that $(p, q) \in \tau_z$. Alternatively, it may return the error symbol $\perp$ if no such q exists or if its internal computation fails. Nevertheless, for each state q such that $(p, q) \in \tau_z$, the procedure may return q. During its computation, the subroutine visits cells of the block containing $\widetilde{x}$ only.

`simulateLeft` starts from and ends in the last position of $\widetilde{x}$ (resp. of $\widetilde{y}$ if $y \neq \varepsilon$) with a state p given as argument and returns a state q such that $(p, q) \in \tau_{zx}$ (resp. $(p, q) \in$

τ_{zxy}). Alternatively, it may return the error symbol $\perp$ if no such q exists or if its internal computation fails. Nevertheless, for each state q such that $(p, q) \in \tau_{zx}$ (resp. $(p, q) \in \tau_{zxy}$), the procedure may return q. During its computation, the subroutine visits cells of the block containing $\widetilde{x}$ (resp. of the portion of the tape containing $\widetilde{xy}$) only.

When the simulated 1-LA is deterministic, it is possible to implement both subroutines in a deterministic way. The implementations of these subroutines are described in Section 6.2.2.3.

6.2.2.2 Main procedure

We now focus on the high-level description of the simulation, which is given in Procedure 9. By using a state component of size 2ℓ, named *relative position* and stored in a global variable relativePosition, $\mathcal{A}'$ can store the exact position of its head relative to the current window. We represent it as a pair (i, S), where $i \in \{ 0, \ldots, \ell - 1 \}$ is the position in the scanned block of length ℓ and $\mathrm{S} \in \{ \mathrm{L}, \mathrm{R} \}$ is equal to L (resp. R) if the head is scanning a position in the left (resp. right) block of the window. We suppose that the component is updated at each head move. Using this component, $\mathcal{A}'$ can avoid moving to the left of the current window. More precisely, from a relative position $(0, \mathrm{L})$ (*i.e.*, the leftmost position covered by the window), when a backward move of $\mathcal{A}$ from p to q has to be simulated, $\mathcal{A}'$ calls the procedure `readFromTable` with argument q, in order to find a state r such that $(q, r) \in \tau_z$, where z is the content of the tape to the left of the window. Hence, it simulates not only the backward step from p to q, but also a complete computation segment to the left of the window, namely, it simulates $z'Xp \vdash z'qX \vdash^* z'Xr$, where $z'X = z$.

In addition to the relative position, $\mathcal{A}'$ stores in a global variable, named relativeFrontier, the relative position of the current frontier, to which we refer as *relative frontier*. Since this position always occurs in the right block of the window, it is enough to represent it as an index $\rho \in \{ 0, \ldots, \ell - 1 \}$. Much like the relative position component, we suppose that it is updated each time a cell is visited for the first time. Observe that such updates

Procedure 9: `main`

```
/* the variables relativePosition and relativeFrontier are not indicated and
   are supposed to be automatically updated;
   in the following, ‘‘current symbol’’ designates the symbol currently
   read by the head                                                        */
frontierState ← q_0
tableState ← ν(relativeFrontier)
while “current symbol” ≠ ◁ do
    simulateLeft(frontierState)
    if frontierState = ⊥ then REJECT
    move the input head leftward until reaching position (ℓ − 1, L)
    simulateLeft(tableState)
    move the input head rightward until reaching position relativeFrontier − 1
    move the input head one cell to the right
    if “current symbol” ≠ ◁ then
        let (q, γ, 0) = selectTransition(frontierState,“current symb”)
        frontierState ← q
        if frontierState = ⊥ then REJECT
        write(γ,tableState)
        tableState ← ν(relativeFrontier)
simulateLeft(frontierState)
if frontierState ∈ F then ACCEPT else REJECT
```

are increments modulo ℓ. Incrementing $\rho = \ell - 1$ means shifting the window by ℓ cells to the right. In particular, this implies to updating the relative position by switching it from $(\ell - 1, \mathrm{R})$ to $(\ell - 1, \mathrm{L})$.

Initially, the head is scanning the left endmarker, which is considered as the left block of the current window. Hence, the initial relative position and relative frontier are $(\ell - 1, \mathrm{L})$ and 0, respectively.

Using both the relative frontier ρ and the relative position (i, S), $\mathcal{A}'$ can ensure that entering a cell for the first time, may be done only once all the information, which is required to determine the symbol to write on the cell, has been gathered. More precisely, when $\mathcal{A}'$ moves its head to the frontier cell (Line 68), it stores a pair (p, q) in its finite control, such that:

- $p \in Q$ is the state entered by $\mathcal{A}$ in the simulated computation, when visiting for the first time the corresponding cell;
- $q \in Q_\perp$ is either $\perp$ or a state such that $(\nu(\rho), q) \in \tau_{zx}$, where zx denotes the content to the left of the right block of the window (*i.e.*, the block of the frontier cell) in the simulated computation.

The states p and q are stored in two variables, respectively named frontierState and tableState, which are updated through two calls to the subroutine `simulateLeft`:

- from one cell to the left of the frontier, to update frontierState (Line 63);
- from the last cell of the preceding block, to update tableState (Line 66).

Once $\mathcal{A}'$ has updated the variables frontierState and tableState, it moves the head to the cell at relative position (ρ, R) (Line 68), and reads the input symbol $\sigma \in \Sigma \cup \{ \lhd \}$ (Line 69). If $\sigma = \lhd$ then $\mathcal{A}'$ enters a final mode in which it calls the subroutine `simulateLeft` with argument frontierState from the current position, and accepts, after violating the endmarker, if the updated value of frontierState is a final state of $\mathcal{A}$ and rejects otherwise (Lines 75 and 76). If $\sigma \neq \lhd$ then $\mathcal{A}'$ simulates a stationary overwriting transition of $\mathcal{A}$ (Lines 70 to 74). Formally, it selects a transition $(p', \gamma, 0) \in \delta(p, \sigma)$, where p is the state stored in frontierState, updates the variable frontierState with p', and overwrites the cell content

with (γ, h) where $h \in T$ is defined as follows according to the value $q \in Q_\perp$ of tableState. If $T = \{0, 1\}$ then $h = 1$ if $\mu(\rho) = (\nu(\rho), q)$ and $h = 0$ otherwise. If $T = Q_\perp$ then $h = q$. It then repeats the procedure with the updated relative frontier. In the case $\rho = \ell - 1$, the window is shifted to the right, in such a way that the head is positioned on the rightmost cell of its left block. This is formally done by setting the relative position to $(\ell - 1, \mathrm{L})$ and the relative frontier to 0.

6.2.2.3 Implementation details for the auxiliary operations and procedures

The operation `selectTransition`

Our subroutines `simulateLeft` and `main` use a basic operation named `selectTransition` (Lines 70 and 81). This operation takes a state $p \in Q$ and a symbol $\sigma \in \Gamma$ as arguments, and returns a tuple $(q, \gamma, d) \in \delta(p, \sigma)$. When no such transition exists, it returns $(\perp, \sigma, 0)$. Notice that the operation is nondeterministic only if $\mathcal{A}$ is nondeterministic.

The operation `write`

In our simulation, $\mathcal{A}'$ overwrites symbols in Σ with symbols in $(\Gamma \setminus \Sigma) \times T$ by performing an operation named `write`. This operation takes two arguments, $\gamma \in \Gamma \setminus \Sigma$ and $r \in Q_\perp$. When $T = \{0, 1\}$, it compares r to the state q such that $\mu(i) = (\nu(i), q)$ where i is the index of the current relative position. If $r = q$ then the symbol $(\gamma, 1)$ is written, otherwise (including the case $r = \perp$) the symbol $(\gamma, 0)$ is written. When $T = Q_\perp$, the routine simply overwrites the content of the currently scanned cell with (γ, r).

The subroutine `readFromTable`

This subroutine was introduced in Section 6.2.2.1. It is used to prevent the head of $\mathcal{A}'$ to move to the portion of the tape on the left of the current window. It is always called from the leftmost position of the window. In particular, this position is the first one of a frozen block $\widetilde{x} \in (\Gamma' \setminus \Sigma)^\ell$, supposed to contain on its second track the encoding of the table τ

that describes the possible computation segments to the left of the window. The routine takes a global variable var as argument, initially containing a state $p \in Q$.

The procedure operates in two modes. First, it moves the head rightward until reaching a position i of $\widetilde{x}$ such that $\nu(i) = p$. Second, it moves the head backward to the first position of $\widetilde{x}$ and halts. When switching from the former to the latter mode at position i, the variable var is updated with an element $q \in Q_\perp$, which is deduced from the scanned symbol $(\gamma, t) \in (\Gamma \setminus \Sigma) \times T$, and the current relative position i, as now explained. If $T = \{0, 1\}$ then, according to whether t equals 0 or 1, q is equal to $\perp$ or is the state such that $\mu(i) = (\nu(i), q)$, respectively. If $T = Q_\perp$ then q is equal to t. In the nondeterministic case, such a position i is nondeterministically chosen. In the deterministic case, however, $\mathcal{A}'$ can select the position i deterministically. Indeed, when $T = Q_\perp$, there exists exactly one i such that $\nu(i) = q$. On the other hand, when $T = \{0, 1\}$, several such indices may exist, but at most one is such that the symbol (γ, t) written at the corresponding position satisfies $t = 1$, by determinism of A. In this latter case, when no such i exist, the procedure sets the variable var to $\perp$. Thus, the routine deterministically finds this position, and returns the image $q \in Q_\perp$ of p by the *functional* table τ written on $\widetilde{x}$.

Lemma 6.2. *The procedure* `readFromTable` *can be implemented using* 2 *states, not counting the global variables* var *and* relativePosition. *Furthermore, the implementation is deterministic whenever $\mathcal{A}$ is deterministic.*

Proof. In all the cases described above, the procedure needs only one state to move the head rightward until finding the correct information, and a second state to move the head back to the initial position, namely to relative position $(0, \mathrm{L})$. The recovered information is directly stored in var. □

The subroutine `simulateLeft`

This subroutine was introduced in Section 6.2.2.1. It is used to update the variables frontierState and tableState before visiting the frontier cell. Hence, the routine has two call-modes:

- one for updating frontierState starting from one cell to the left of the frontier;
- the other one for updating tableState starting from the rightmost cell of the left block of the window.

Let us denote by var the variable to be updated and by (i, S) the relative position from which the routine is called. Let $zxy \in \rhd(\Gamma \setminus \Sigma)^*$ be the projection on $(\Gamma \setminus \Sigma) \cup \{ \rhd \}$ of the tape content up to the starting position, with x corresponding to the left block of the current window. During the computation, `simulateLeft` has access to the content of the window up to position (i, S). It basically performs a direct simulation of $\mathcal{A}$ on the corresponding part of the tape, and uses the procedure `readFromTable` in order to simulate computations that occur to the left of the window, as explained above. Moreover, if a rightward transition $(q, \gamma, +1) \in \delta(r, \gamma)$ from the last position of zxy has to be simulated, then the procedure halts without performing the right move, namely at relative position (i, S), and updates the variable var to the value q. Notice that the direct simulation is deterministic if $\mathcal{A}$ is deterministic, since the procedure `readFromTable` is deterministic in this case, by Lemma 6.2.

However, this naive approach might fail because the direct simulation may enter loops and never halt. In order to handle this issue, we need to detect computational loops. We proceed as follows. If $(p, q) \in \tau_{zxy}$ then this can be witnessed by a direct simulation which never repeats a configuration. In particular, the same state cannot be entered twice at the same position. Since the procedure `simulateLeft` operates in a read-only window of size at most 2ℓ, any repetition-free computation has length bounded by $2\ell n$ (counting the calls to the subroutine `readFromTable` as a single move). Hence, by using a clock of size $2\ell n$, stored in the finite control of the simulating device, we can enforce the procedure to halt. Only runs that halted before this time limit may return a state while "killed" runs will return $\perp$. Notice that the clock yields a polynomial increase of the size of the simulating machine only.

Let us focus on the implementation details of `simulateLeft` given in Procedure 10. The subroutine starts by storing the value (i, S) of the current relative position (Line 77)

Procedure 10: `simulateLeft(var)`

```
/* the variables relativePosition and relativeFrontier are not indicated and
   are supposed to be automatically updated                                  */
```

Input: a variable var in read/write mode, initially containing a state in Q

Output: halts with var containing a state or the special symbol $\perp$

77 let $(i, \mathrm{s}) =$ relativePosition

78 clock $\leftarrow 2\ell n$

79 **while** var $\neq \perp$ **and** clock > 0 **do**

80 let $\gamma \in \Gamma_{\triangleright\triangleleft} \setminus \Sigma$ be the first track symbol of the currently scanned cell

81 let $(q, \gamma, d) =$ `selectTransition`(var, γ)

82 var $\leftarrow q$

83 **if** var $= \perp$ **then**

84 **break**

85 **else if** relativePosition $= (i, \mathrm{S})$ **and** $d = +1$ **then**

86 **break**

87 **else if** relativePosition $= (0, \mathrm{L})$ **and** $d = -1$ **then**

88 `readFromTable(var)`

89 **else**

90 move head according to d

91 clock $\leftarrow$ clock $- 1$

92 **if** clock $= 0$ **then** var $\leftarrow \perp$

93 move head rightward until reaching position (i, S)

and the clock to $2\ell n$ (Line 78). After that, $\mathcal{A}'$ simulates $\mathcal{A}$ by reading the first track of the scanned cell and using the transition function of the simulated machine (Lines 80 and 81). The next simulated state reached by $\mathcal{A}$ is stored in the variable var (Line 82). If, by incompleteness of the transition function of $\mathcal{A}$ no such state exists, var gets value $\perp$ and the procedure ends after moving its head rightward to the position (i, S) from which it was called (Lines 83, 84 and 93). In case a right move from relative position (i, S) is detected, the procedure ends without moving rightward (Lines 85 and 86). If a left move from the relative position $(0, \mathrm{L})$ is detected, $\mathcal{A}'$ calls `readFromTable` in order to simulate the computation segment to the left of the window (Lines 87 and 88). Otherwise, $\mathcal{A}'$ can directly simulate the transition performed by $\mathcal{A}$: the simulating machine moves its head according to the simulated transition (Line 90). After each simulated move[3] of $\mathcal{A}$, the value of the clock is decremented (Line 91). If after $2\ell n$ simulated moves the routine has not halted, then it updates the value of var with $\perp$, moves the head rightward until reaching the relative position (i, S), and halts (Lines 92 and 93).

Hence, implementing `simulateLeft` requires only a polynomial number of states in n.

Lemma 6.3. *The subroutine* `simulateLeft` *can be implemented using* $12\ell n + 2$ *states, not counting the global variables* var, relativePosition, *and* relativeFrontier. *Furthermore, it is deterministic if* $\mathcal{A}$ *is deterministic.*

Proof. Using Lemma 6.2, the implementation of `simulateLeft` given in Procedure 10 is deterministic when $\mathcal{A}$ is deterministic. Furthermore, it uses the following state components.

- A binary state component is required for keeping track of the call-mode in which the procedure is operating. Moreover, since the mode also determines the position from which the procedure is called, and because the relative frontier (stored in the variable relativeFrontier) is not modified during the execution of the procedure, no

[3]Here, we consider the simulated computation segments to the left of the window, which are recovered through call to `readFromTable` (Line 88), as single moves.

further state components are necessary for storing the initial value (i, S) of relativePosition.

- A state component of size $2\ell n$ is required for storing the value of the clock (we do not need to store the value 0, as the device can directly enter a failure state when decrementing the clock from value 1).
- A state component of size 3 storing the three internal modes of the procedure, namely the main mode and the two sub-modes resulting from the calls to the routine `readFromTable` (Line 88) by Lemma 6.2.
- One failure state for each call-mode is required for moving the head rightward until reaching the initial position (i, S) when the procedure fails (Line 93).

Hence, the total number of states required for implementing `simulateLeft` is $2 \cdot (2\ell n \cdot 3 + 1)$, not counting the global variables var, relativePosition, and relativeFrontier. □

6.2.2.4 Sipser's simulation

When the simulated 1-LA $\mathcal{A}$ is deterministic it is possible to use a finer implementation of `simulateLeft`, thus avoiding the size-expensive clock. This finer implementation is an adaptation of a construction due to Sipser, that avoids deterministic loops in deterministic Turing machines by a clever backward simulation [Sip80]. A version for 2DFAs has been presented by Geffert, Mereghetti, and Pighizzini, who showed that a linear increase of the size is sufficient for simulating any 2DFA with a halting equivalent one [GMP07]. We first recall the main ideas of this simulation and then we show how it can be adapted for implementing the procedure `simulateLeft`.

Let $\mathcal{B}$ be a 2DFA. Without loss of generality (see [GMP07, Lemma 3.1]), we can suppose that $\mathcal{B}$ cannot perform stationary moves, that it has exactly one final state q_F, that acceptance is made by entering q_F at the leftmost position, namely on the left endmarker, and that furthermore no transition can be performed from that point, *i.e.*, from state q_F

reading $\triangleright$. Then, given an input word w, we consider the *configuration graph* $\mathcal{G}$ of $\mathcal{B}$ on w, namely the directed graph such that the vertices are the configurations of $\mathcal{B}$ on w and there is an edge from c to c' if $c \vdash c'$. In particular, w is accepted by $\mathcal{B}$ if and only if there exists a path in $\mathcal{G}$ from the initial configuration c_I to the unique final configuration c_F. Let us focus on the connected component of c_F. Since $\mathcal{B}$ is deterministic and because c_F has no successor by assumption, this component is a tree rooted in c_F. Moreover, w is accepted if and only if c_I occurs in this tree. Hence, by performing a depth-first-search in the tree, one can check whether c_I is in the component, and thus decide whether the word w is accepted. This depth-first-search idea can be implemented deterministically, starting from c_F, using four copies of each state only. This yields a $(4 \cdot \#Q_{\mathcal{B}})$-state halting 2DFA which is equivalent to $\mathcal{B}$, where $Q_{\mathcal{B}}$ denotes the state set of $\mathcal{B}$ [GMP07]. It should be noticed that when c_I does not belong to the tree, the simulating halting 2DFA halts in a configuration that matches the root c_F after having tried all its subtrees.

We shall adapt this construction to our case, for implementing `simulateLeft`. Notice that the direct simulation of $\mathcal{A}$ on the corresponding frozen portion of the tape is a read-only deterministic computation. Remember that `simulateLeft` can operate in two call-modes. Let us fix one of these two modes. We denote by var the variable to update, by p its initial content, by (i, S) the starting relative position, and by ρ the relative frontier. Let zwX be the content of the tape to the left of the starting position, with z being the portion to the left of the window and $|X| = 1$.

We consider a 2DFA variant $\mathcal{C}$ simulating $\mathcal{A}$, which starts from and ends in some specified position. We assume that it has access to the variables relativePosition and relativeFrontier. It basically performs an unclocked version of Procedure 10 in which the failure state (Line 93) has been thrown away. Furthermore, we ignore the last update of var, when the image q of p by τ_{zwX} is found. We can suppose that $\mathcal{C}$ never performs stationary moves. Indeed, on the one hand, $\mathcal{A}$ does not perform any stationary move when working on frozen symbols by assumption. On the other hand, without increasing the size of the device, we can eliminate the stationary moves possibly resulting from the

calls to the subroutine `readFromTable`, by using classical techniques. Moreover, $\mathcal{C}$ can be implemented using 3 states only, not counting the variable var, according to Lemma 6.2 (one state for the main mode, and two additional states for the calls to the sub-routine `readFromTable`). Remember that X is the symbol written at relative position (i, S) and let $R(X) = \{ r \mid \exists q, \delta(r, X) = \{ (q, X, +1) \} \}$. An *accepting configuration* of $\mathcal{C}$, is a configuration in which the head is positioned at relative position (i, S) (thus scanning X), the internal state corresponds to its main mode (*i.e.*, not to a call to `readFromTable`), and var contains a state $r \in R(X)$.

Observe that, although we dropped the last update of var with respect to Procedure 10, it is possible to recover it from the halting point of $\mathcal{C}$. Indeed, this value is either $q \in Q$ if $\mathcal{C}$ halted in an accepting configuration with var storing $r \in R(X)$ such that $\delta(r, X) = \{ q, X, +1 \}$, or $\perp$ otherwise. Notice that $\mathcal{C}$ does not use the endmarkers but rather the relative position, to ensure that the head stays between $(0, \mathrm{L})$ and (i, S). In particular, the *initial configuration* c_I of $\mathcal{C}$ is at relative position (i, S) with the state corresponding to its main mode and var storing p.

The main issue for adapting Sipser's construction to our case, is that the target configuration c_F is initially unknown. Indeed, it is the role of `simulateLeft` to find the image q of p by τ_{zwX} (when defined). In order to solve this issue, we apply the Sipser simulation of $\mathcal{C}$ for each value $r \in R(X)$ of var. By taking such r's in order, we only need 4 internal sub-modes for the simulation, as in [GMP07]. Indeed, if for some target configuration $c_F(r)$ the simulation does not find c_I, then it halts in a configuration that encodes $c_F(r)$. In particular, var contains r and can thus be updated with the next value from $R(X)$. If no successor of r exists, then $\mathcal{C}$ can never reach an accepting configuration, namely c_I is in none of the trees rooted in the $c_F(r)$'s. Thus, our procedure updates var with the symbol $\perp$ and halts. If otherwise the configuration c_I has been found during a simulation starting from some $c_F(r)$, then the direct execution of $\mathcal{C}$ halts in $c_F(r)$. Thus, it is enough to directly simulate $\mathcal{C}$ from c_I and to update var according to the state r which is recovered when the execution halts. More precisely, once $\mathcal{C}$ has reached $c_F(r)$, our proce-

dure reads the symbol X and updates the variable var which contains $r \in R(X)$ with the state q such that $\delta(r, X) = \{ (q, X, +1) \}$.

We now evaluate the size of this deterministic version of `simulateLeft`, which works independently of the chosen table encoding (*i.e.*, for any T, ℓ), so long as $\mathcal{A}$ is deterministic.

Lemma 6.4. *If $\mathcal{A}$ is deterministic, then, independently of the encoding of the Shepherdson tables on the tape, the procedure* `simulateLeft` *can be implemented deterministically using* 30 *internal states, not counting the global variables* var, relativePosition, *and* relativeFrontier.

Proof. We evaluate the implementation described above. As in the implementation given in Section 6.2.2.3, we need a binary component for storing the call-mode of the procedure. This component is sufficient to recover the starting position (i, S), possibly by reading the value of relativeFrontier which is not modified during the execution of the procedure.

Then the 3-mode automaton $\mathcal{C}$ given above is simulated a first time, by Sipser's construction, using 4 sub-modes. If the simulation succeeds then $\mathcal{C}$ is simulated a second time in order to recover the value to store in var. Otherwise, no further state is needed, as a failure necessarily occurs at position (i, S) from which var is updated to $\perp$ without moving the head.

Thus, not counting the global variables relativePosition, relativeFrontier, frontierState, and tableState, we obtain that $2(12 + 3)$ states are enough. □

6.2.3 Properties of $\mathcal{A}'$

In this section, we state the main properties of the 1-LA $\mathcal{A}'$ that has been obtained from $\mathcal{A}$ by the simulation defined in Section 6.2.2. Notice that several simulations have been described. We differentiate them only when required.

Lemma 6.5. *$\mathcal{A}'$ is equivalent to $\mathcal{A}$.*

Proof. We first prove that $L(\mathcal{A}) \subseteq L(\mathcal{A}')$. Consider an accepting computation $\mathbf{c} = c_0, c_1, \ldots, c_t$ of $\mathcal{A}$ on some input $w = w_1 \cdots w_m \in \Sigma^*$, where $c_0 = q_0 \triangleright w \triangleleft$ and $c_t = \triangleright x \triangleleft q_F$, for some

$q_F \in F$ and $x = x_1 \cdots x_m \in (\Gamma \setminus \Sigma)^*$. We can extract from $\mathbf{c}$ the sequence $c_{j_1}, \ldots, c_{j_m}$ of configurations in which the head is scanning a symbol of Σ, namely, for each $i \in \{1, \ldots, m\}$, $c_{j_i} = \triangleright x_1 \cdots x_{i-1} p_i w_i \cdots w_m \triangleleft$ for some state p_i. Notice that p_i is necessarily the first state entered at position i in $\mathbf{c}$. Moreover, since by assumption $\mathcal{A}$ has the form given in Lemma 6.1, we have $c_{j_i+1} = \triangleright x_1 \cdots x_{i-1} p'_i x_i w_{i+1} \cdots w_m \triangleleft$ for some state p'_i. In particular, $(p'_i, x_i, 0) \in \delta(p_i, w_i)$ is the stationary transition which is performed when overwriting the content of the cell at position i during the computation $\mathbf{c}$. For convenience, we define $c_{j_{m+1}}$ to be the first configuration of the form $\triangleright x_1 \cdots x_m p_{m+1} \triangleleft$ for some state p_{m+1}. We also set $x_0 = \triangleright$, $x_{m+1} = \triangleleft$, $p_0 = p'_0 = q_0$, $p'_{m+1} = p_{m+1}$, and $p_{m+2} = q_F$. For each $i \in \{0, \ldots, m+1\}$, we have $(p'_i, p_{i+1}) \in \tau_{x_0 \cdots x_i}$.

The simulating 1-LA $\mathcal{A}'$ recovers $\mathbf{c}$ by successively storing the states

$$p'_0, p_1, p'_1, \ldots, p_{m+1}, p'_{m+1}, p_{m+2}$$

in its internal variable frontierState, while visiting the corresponding cells from left to right. More precisely, for each $i \in \{0, \ldots, m+2\}$, when $\mathcal{A}'$ enters the i-th tape cell for the first time, the variable frontierState contains the state p_i. It is routine to show by induction that, at each iteration of the while loop in Procedure 9 (Lines 62 to 74), as soon as the left block of the current window encodes the correct table, the two calls to the subroutine `simulateLeft` can recover the right information, namely frontierState is updated from p'_i to p_{i+1} on Line 63, and tableState is updated from $\nu(\rho)$ to q on Line 66, where q is such that:

- if $T = \{0, 1\}$ then either q is a state r such that $(\nu(\rho), r) \in \tau_{x_0 \cdots x_k}$, or $q = \bot$;
- if $T = Q_\bot$ then q is the image of $\nu(\rho)$ by $\tau_{x_0 \cdots x_k}$ if defined, or $\bot$ otherwise;

where k is the rightmost position of the left block of the window.

Conversely, updating frontierState in $\mathcal{A}'$ is done only by performing direct simulation of $\mathcal{A}$ that may read some table τ, which have previously be written on the frozen content of the tape. By induction on the frontier position, we can prove that τ is a

relation included in $\tau_{x_0 \cdots x_{k-\ell}}$, where k is the rightmost position of the left block of the window. (We further have $\tau = \tau_{x_0 \cdots x_{k-\ell}}$ when $\mathcal{A}$ is deterministic.) Hence, every state recovered through `readFromTable` and `simulateLeft`, correspond to a state that can be entered by $\mathcal{A}$ from the corresponding configuration at the corresponding position. Thus, for each accepting computation of $\mathcal{A}'$, one can find a simulated accepting computation of $\mathcal{A}$, whence $L(\mathcal{A}') \subseteq L(\mathcal{A})$. □

We have shown how $\mathcal{A}'$ simulates $\mathcal{A}$ in a halting manner, by shifting a virtual window to the right during its computation, and by restricting local head moves to the current window of size 2ℓ. We now evaluate the size of $\mathcal{A}'$. We point out that, as long as $\mathcal{A}$ is deterministic, the two possible encodings of the Shepherdson tables, namely using $T = \{0, 1\}$ and $\ell = n^2$ or using $T = Q_\perp$ and $\ell = n$, are possible. For both, the Sipser simulation yields a smaller size increase with respect to the clock trick used for the general case. Though in the deterministic case, the smallest simulating 1-LA is obtained by combining the second encoding with Sipser simulation, it should be noticed that using the first encoding yields a smaller working alphabet, whose size does not depend on n.

Lemma 6.6. *$\mathcal{A}'$ has polynomial size with respect to $\mathcal{A}$. More precisely, we obtain the following simulation costs:*

case	***technique***	***states***	***working symbols***
nondeterministic	*clock*/$\ell = n^2$	$O(n^9)$	$2 \cdot \#(\Gamma \setminus \Sigma)$
deterministic	*Sipser*/$\ell = n^2$	$O(n^6)$	$2 \cdot \#(\Gamma \setminus \Sigma)$
	Sipser/$\ell = n$	$O(n^4)$	$(n+1) \cdot \#(\Gamma \setminus \Sigma)$

Proof. The set of working symbols of $\mathcal{A}'$ is $\Gamma' \setminus \Sigma = (\Gamma \setminus \Sigma) \times T$. In both nondeterministic and deterministic cases, the finite control uses several components:

- the variable relativeFrontier of size ℓ;
- the variable relativePosition of size $2\ell - 1$ (the value $(\ell - 1, \mathrm{R})$ is never used, since the relative position is always to the left of the relative frontier);
- the variable frontierState of size n (the value $\perp$ is unnecessary, since updating frontierState with $\perp$ implies rejecting the input);

- the variable tableState of size $n + 1$;
- the state components used to implement `simulateLeft` of size:
  - $12\ell n + 2$ using the clock (general case, see Section 6.2.2.3);
  - 30 using Sipser's construction (deterministic case only, see Section 6.2.2.4).

  In both cases, the size includes the two sub-modes used in the implementation of the routine `readFromTable`, *cf.* Lemma 6.2.

□

Let us now evaluate the time used by the simulating machine $\mathcal{A}'$.

Lemma 6.7. *In every computation of $\mathcal{A}'$, each tape cell is visited a number of times which is bounded by some polynomial in the size of $\mathcal{A}$.*

Proof. Let us fix a cell c. As $\mathcal{A}'$ is loop-free, each time the head visits c it must have a different state or a different tape content. A tape modification between two visits of c is restricted to cells from the right block of the current window containing c. The number of successive tape modifications in a window is bounded by ℓ. Indeed, after ℓ overwritings the window is shifted. The cell c may occur in two successive windows: first in the right part and, after shifting the window, in the left part. Thus, the number of visits to the cell c is bounded by $2\ell n'$, where n' is the number of states of $\mathcal{A}'$, which is polynomial in the number of states of $\mathcal{A}$ as seen in Lemma 6.6. The number of visits to each cell is hence bounded by a polynomial in the size of $\mathcal{A}$. □

As a consequence, $\mathcal{A}'$ operates in linear time with respect to the input length.

We can observe that, by the use of the state components relativePosition and relativeFrontier, our simulating 1-LA always "knows" where the frontier is. Roughly, this means that $\mathcal{A}'$ does not use the meta-instruction "move rightward until finding the leftmost cell that has not been visited so far" which was used by the 1-LA described in Example 6.1.

Lemma 6.8. *$\mathcal{A}'$ "knows" where the frontier is, namely, there exist special states that are entered exactly when visiting a tape cell for the first time.*

Proof. In Procedure 9, when entering the frontier cell (Line 68), the simulating device enters a particular mode from which the cell is scanned (Line 69) in order to simulate a stationary overwriting transition of $\mathcal{A}$ (Lines 70 to 74). Hence, we can exhibit the set of states corresponding to this special mode. □

6.3 Main Result and Consequences

We are now able to state our results as consequences of the properties of $\mathcal{A}'$ stated in Section 6.2.3. See Figure 6.1 for a summary of these results.

6.3.1 Main result: Conversion into linear-time 1-limited automata

Our main result shows that operating in super-linear time is not essential for 1-LAs, if allowing a polynomial increase in the number of states.

Theorem 6.1. *Each* 1-LA *(resp. deterministic* 1-LA*) admits an equivalent linear-time* 1-LA *(resp. deterministic* 1-LA*) of polynomially larger size.*

Proof. We start with a 1-LA. By paying a linear increase of its size and preserving determinism, we transform it into an equivalent 1-LA $\mathcal{A}$ which performs stationary moves exactly when rewriting a cell content, by Lemma 6.1. Then we apply the above construction in order to obtain the 1-LA $\mathcal{A}'$ equivalent to $\mathcal{A}$, by Lemma 6.5. If $\mathcal{A}$ is deterministic then so is $\mathcal{A}'$. By Lemma 6.6, the size of $\mathcal{A}'$ is bounded by some polynomial in the size of $\mathcal{A}$. By Lemma 6.7, $\mathcal{A}'$ operates in linear time in the length of the input. □

6.3.2 Conversion into weight-reducing Hennie machines

Linear-time 1-LAs are particular cases of Hennie machines (*i.e.*, linear-time linear bounded automata), hence, it follows from the above result that any 1-LA can be transformed into a Hennie machine with a polynomial increase of the size only. Using Lemma 6.7, we can ac-

tually obtain the stronger result that the 1-LA can be transformed into a weight-reducing Hennie machine of polynomial size.

Theorem 6.2. *Each* 1-LA *(resp. deterministic* 1-LA*) admits an equivalent weight-reducing Hennie machine (resp. deterministic weight-reducing Hennie machine) of polynomial size.*

Proof. Following [Prů14, Lemma 4], it is enough to modify the 1-LA $\mathcal{A}'$ obtained by the above construction, in such a way that each time a frozen cell is visited, it is overwritten with a copy of the frozen symbol, that encodes the number of visits to the cell. Since, by Lemma 6.7, the total number of visits of a cell in a computation of $\mathcal{A}'$ is bounded by some polynomial in the size of $\mathcal{A}$, the transformation yields an equivalent weight-reducing Hennie machine which has polynomial size with respect to the simulated 1-LA. Furthermore, the conversion clearly preserves determinism. □

6.3.3 Conversion into two-way automata with common guess

Some 1-LAs have a particular behavior, which can be decomposed into two phases. In the first phase, they nondeterministically rewrite the content of the whole tape during a left-to-right traversal of the input. Then, in the second phase, they perform a two-way read-only computation over the overwritten tape. To formally define this kind of 1-LAs, we introduce the following model.

Definition 6.1. *A* 2NFA *(resp.* 2DFA*)* with common guess *(*2NFA+cg*, resp.* 2DFA+cg*)*[4] *is a tuple* $\langle \mathcal{A}, \Sigma, \Delta \rangle$ *where* Σ *and* Δ *are two alphabets and* $\mathcal{A}$ *is a* 2NFA *(resp.* 2DFA*) over the product alphabet* $\Sigma \times \Delta$*.*

The model is aimed to recognize languages over Σ. Its dynamics is defined as for two-way automata, but a nondeterministic pre-computation initially marks each input symbol with a symbol from Δ.[5] Hence, the read-only automaton $\mathcal{A}$ has access to both

[4] 2DFA+cgs also correspond to *synchronous two-way deterministic finite verifiers* [Kap14a].

[5] Though the model is motivated by special behaviors of 1-LAs whence the nondeterministic pre-computation is naturally thought as being one-way left-to-right, there is no reason to impose this. Here,

the input symbol and the guessed additional information. The language accepted, denoted $L(\langle \mathcal{A}, \Sigma, \Delta \rangle)$, is defined as the projection, denoted π_1, of $L(\mathcal{A})$ on the alphabet Σ, *i.e.*, $L(\langle \mathcal{A}, \Sigma, \Delta \rangle) = \pi_1(L(\mathcal{A}))$. In other terms, a word is accepted by $\langle \mathcal{A}, \Sigma, \Delta \rangle$ if for some guess, the enriched word in $(\Sigma \times \Delta)^*$ is accepted by $\mathcal{A}$. We point out that, due to the common guess, 2DFA+cg's are nondeterministic devices.

Let us detail the connection between 1-LAs and 2FA+cgs. It is easy to turn a 2FA+cg into an equivalent nondeterministic 1-LA of the same size, by simply guessing and writing the symbols from Δ when visiting the cells for the first time. It is however *a priori* not clear whether a converse transformation with reasonable size cost exists. The main issue for such a conversion is that, at any time during a computation of a 1-LA, a position of the tape is identified as being the leftmost cell that has not been visited so far, namely the current frontier. In particular, a 1-LA can use meta-instructions making use of this identified position, such as "move the head rightward to the frontier cell", as it is the case in Example 6.1. Nevertheless, when a 1-LA does not use such kind of instructions, that is, if it always "knows" when it enters a cell for the first time (before scanning its content), then it is easy to convert it to an equivalent 2FA+cg of similar size. Formally, the property of always "knowing" where the frontier is can be expressed by specifying the states that are entered exactly when visiting a tape cell for the first time. In order to simulate such a 1-LA (with input alphabet Σ and working alphabet Γ) with a 2FA+cg (with common guess alphabet Δ), it is indeed enough to first guess the working symbols that will be written on the tape at the end of the computation (thus, setting $\Delta = \Gamma$), and then simulate the 1-LA in a read-only manner, using the symbol component in Σ when the cell is visited for the first time (which is determined by the current state) or the symbol component in Γ otherwise, while checking that the guessed symbols correspond to the symbol overwritten during the simulated computation. Notice that, so obtained, the resulting device is a 2DFA+cg

the marking is uniform meaning that each cell is independently marked by a nondeterministically-chosen symbol of Δ. This operation is connected with existential set quantification in *monadic second order logic*, see, *e.g.*, [Boj+17].

(resp. is halting) when the source 1-LA is deterministic (resp. halting).

Concerning the conversion of arbitrary 1-LAs (*i.e.*, not "knowing" where the frontier is) into equivalent 2FA+cgs, it is a non-trivial consequence of our main construction that with a polynomial increase of the size only, this can be achieved.

Theorem 6.3. *Each* 1-LA *(resp. deterministic* 1-LA*) admits an equivalent halting* 2NFA+cg *(resp.* 2DFA+cg*) of polynomial size.*

Proof. By Lemma 6.8, $\mathcal{A}'$ "knows where the frontier is". Hence, by applying the above-given conversion, we can obtain an equivalent 2FA+cg of polynomial size, which is halting by Lemma 6.7. Furthermore, if $\mathcal{A}'$ is deterministic, then the resulting device is a 2DFA+cg. □

In the nondeterministic case, this last result is of particular interest. Indeed, 2NFA+cg's can be seen as particular cases of 1-LAs. (It is not the case for 2DFA+cg's with respect to deterministic 1-LAs.) Hence, Theorem 6.3 gives a kind of normal form for nondeterministic 1-LAs. In particular, it is easy to modify such a 1-LA in order to recognize the reverse of its accepted language.

Corollary 6.1. *Each* 1-LA $\mathcal{A}$ *can be transformed into a nondeterministic* 1-LA $\mathcal{A}'$ *with a polynomial increase of the size, such that* $L(\mathcal{A}') = L(\mathcal{A})^{\mathrm{R}}$.

Proof. Given a 1-LA $\mathcal{A}$, we can obtain an equivalent 2NFA+cg by Theorem 6.3. By replacing left move by right move and *vice versa* on each transition of its underlying automaton, we can obtain a 2NFA+cg of same size, which recognizes $L(\mathcal{A})^{\mathrm{R}}$. This 2NFA+cg can in turn be viewed as a nondeterministic 1-LA. □

6.3.4 Lower bounds

Concerning the size cost of the simulation of 2DFA+cg by deterministic 1-LA, using the language L_n from Example 6.1, we can prove an exponential gap in the deterministic case.

Theorem 6.4. *Let L_n be the language of Example 6.1. Hence*

$$L_n{}^{\mathrm{R}} = \{\, x_k x_{k-1} \cdots x_0 \mid k \in \mathbb{N}, \textit{ for each } i\colon x_i \in \{\, a, b \,\}^n, \textit{ for some } j \neq 0\colon x_j = x_0 \,\}.$$

Then,

1. *$L_n{}^{\mathrm{R}}$ is accepted by a* 2DFA+cg*, a linear-time nondeterministic* 1-LA*, or a deterministic weight-reducing Hennie machine of size polynomial in n;*
2. *any* 1DFA *recognizing $L_n{}^{\mathrm{R}}$ requires 2^{2^n} states;*
3. *any deterministic* 1-LA *recognizing $L_n{}^{\mathrm{R}}$ requires $O(2^n)$ states.*

Proof. Example 6.1 describes a deterministic 1-LA recognizing L_n, whose size is linear in n. By applying Theorems 6.2 and 6.3, we respectively obtain equivalent deterministic weight-reducing Hennie machine and 2DFA+cg of polynomial size. Both models can be transformed with at most a linear increase in size, in order to accept the reverse of the language, thus proving Item 1. For both models, it is indeed enough to initially move the head to the right endmarker, and then simulate the two-way device in opposite direction, that is, replacing left moves of the head by right ones and *vice versa*. In the case of 2DFA+cg this yields a constant increase of the size of the model (only one state should be added for the initial mode). In the case of weight-reducing Hennie machines, since during the initial traversal of the input the cells should be overwritten in a decreasing way (in order to preserve the weight-reducingness property), we should in addition add a fresh copy of each input symbol to the set of working symbols.

Using a simple distinguishability argument, we can prove Item 2. Finally, Item 3 can be deduced from this previous point and the exponential upper bound for the size cost of the simulation of deterministic 1-LA by 1DFA given in [PP14]. □

7

Two-Way Automata and One-Tape Machines

In this chapter we continue the investigation about devices operating in linear time. In particular, we shall compare the sizes of descriptions of finite automata with those of equivalent one-tape Turing machines working in linear time. Throughout the chapter we consider the variants of one-tape deterministic Turing machines introduced in Section 2.2.1, that we recall in Figure 7.1.

It is useful to emphasize that, as we shall prove, it cannot be decided whether or not a one-tape Turing machine works in linear time.[1] Furthermore, there is no recursive function bounding the size blowup from one-tape Turing machines working in linear time to equivalent finite automata. These results remain true in the restricted case of bounded machines.

To overcome the above-mentioned "negative" results, we consider weight-reducing machines, that can be seen as a syntactical restriction on one-tape Turing machines. We focus on the deterministic case. These devices can have non-halting computations. However, they work in linear time as soon as they are halting. In fact, we show that it is possible to decide whether a weight-reducing machine is halting. As a consequence, it is also possible to decide whether it works in linear time. Furthermore, with a polynomial size increase, any such machine can be made halting whence working in linear time. A

[1] For the sake of completeness, we mention that it is decidable whether or not a machine makes at most $cm + d$ steps on input of length m, for any *fixed* $c, d > 0$ [Gaj15].

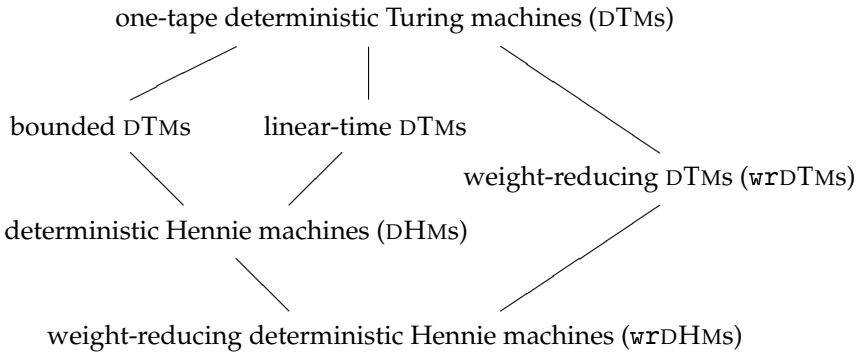

Figure 7.1: Variants of one-tape deterministic Turing machines.

double exponential blowup to 1DFAs is proved.

The same blowup is easily extended to weight-reducing Hennie machines.

The first part of the chapter is devoted to investigate these and some other properties of these models. These results are interesting *per se*. Moreover, some of them turn out to be useful in the second part of the chapter, where we relate the study of these restricted variants of one-tape machines, accepting only regular languages, to the famous Sakoda and Sipser question concerning the size blowups from 1NFAs or 2NFAs to 2DFAs.

Here, we study blowups for the conversion of 1NFAs and 2NFAs into several variants of linear-time one-tape deterministic Turing machines.

Our main result is that each 2NFA $\mathcal{A}$ can be simulated by a one-tape deterministic Turing machine which works in linear time (with respect to the input length) and which has a polynomial size with respect to the size of $\mathcal{A}$. We point out that the resulting machine can use extra space, besides the tape segment which initially contained the input. Next, the machine is halting and weight reducing, thus implying a linear execution time. Hence, nondeterminism can be eliminated with at most a polynomial size increase, obtaining a linear execution time in the input length, and provided the ability to rewrite tape cells and to use some extra space.

We then investigate what happens by removing the latter possibility, namely if the machine does not have any further tape storage, *i.e.*, it is a Hennie machine. We prove that even under this restriction it is still possible to obtain a machine of polynomial size,

namely each 2NFA can be transformed into an equivalent Hennie machine of polynomial size. However, the machine resulting from our construction is not weight reducing, unless we require that it agrees with the given 2NFA only on sufficiently long inputs. We do not have this problem in the unary case, namely for a one-letter input alphabet, where we prove that each unary 2NFA can be simulated by a weight-reducing Hennie machine of polynomial size. Similar results are obtained for the transformation of 1NFAs into variants of one-tape deterministic machines.

The chapter is organized as follows. Section 7.1 is devoted to prove the above mentioned undecidability and non-recursive trade-off results.

Sections 7.2 and 7.3 are devoted to study some fundamental properties of weight-reducing machines. In Section 7.2, after proving that it can be decided if a deterministic Turing machine is weight-reducing, we show that each linear-time machine $\mathcal{T}$ can be turned into an equivalent weight-reducing one whose size is bounded by a function of the size and of the execution time of $\mathcal{T}$, and we present a simulation of weight-reducing machines by finite automata, studying its size cost. In Section 7.3 we show how to decide if a weight-reducing machine halts on any input and if it works in linear time. As a consequence of this result, we are also able to prove that by a polynomial size increase, each weight-reducing machine can be transformed into and equivalent one which always halts and which works in linear time.

In Section 7.4 we present our main simulation result: we show that each n-state 2NFA can be transformed into an equivalent halting weight-reducing machine of size polynomial in n. In Section 7.5 we discuss how the simulation changes if the resulting machine is required to be a Hennie machine. Finally, in Section 7.6 we revise the results of Sections 7.4 and 7.5 under the assumption that the simulated automata are *one-way* instead of two-way.

The results shown in this chapter have been presented in [Gui+18].

7.1 Hennie Machines: Undecidability and Non-Recursive Trade-Offs

In this section we investigate some basic properties of DTMs. First of all, we prove that it cannot be decided whether or not a bounded DTM works in linear time. As a consequence, it cannot be decided if a DTM is a Hennie machine. Since linear-time DTMs accept only regular languages [Hen65], it is natural to investigate the size cost of their conversion into equivalent finite automata. Even in the restricted case of deterministic Hennie machines we obtain a "negative" result, by proving a non-recursive trade-off.

Let us start by proving the following undecidability result.

Theorem 7.1. *It is undecidable whether a bounded* DTM *works in a linear time.*

Proof. We show that the problem of deciding whether a DTM halts on the empty word ε reduces to this problem. Let $\mathcal{T} = \langle Q, \Sigma, \Gamma, \delta, q_0, F \rangle$ be a DTM. Without loss of generality, assume that $\mathcal{T}$ has a tape infinite only to the right. Construct a bounded Turing machine $\mathcal{H}$ with the input alphabet $\{ a \}$ as follows. Given an input $v \in a^*$, $\mathcal{H}$ starts to simulate $\mathcal{T}$ over ε. If the simulation reaches the $|v| + 1$-st tape cell, then $\mathcal{H}$ stops the simulation and performs additional $\Theta(|v|^2)$ computation steps over the first $|v|$ tape cells. Otherwise, $\mathcal{H}$ continues the simulation of $\mathcal{T}$ and halts, if $\mathcal{T}$ halts. One can verify that the construction yields the following properties.

- If $\mathcal{T}$ halts on ε in time t visiting s tape cells, then $\mathcal{H}$ performs $O(t)$ computation steps on any input of length greater than s, while it performs $O(|s|^2)$ steps on shorter inputs. In both cases, the time is constant in the input length.

- If $\mathcal{T}$ does not halt on ε, then for any input v either the simulation reaches the $|v| + 1$-st tape cell and then $\mathcal{H}$ performs further $\Theta(|v|^2)$ computation steps, or it does not halt because $\mathcal{T}$ enters an infinite loop, without reaching such a tape cell. In both cases $\mathcal{H}$ is not a linear-time DTM.

This allows to conclude that $\mathcal{H}$ is a linear-time DTM if and only if $\mathcal{T}$ halts on input ε, which is known to be undecidable. □

We now show that the size trade-off from linear-time DTM to finite automata is not recursive. More precisely, we obtain a non-recursive trade-off from Hennie machines to finite automata.

Theorem 7.2. *There is no recursive function bounding the size blowup when transforming a* DHM *to finite automata.*

Proof. For each $n > 0$, let w_n be the string over $\{ a \}$ of length $\Sigma(n)$, where Σ is the busy beaver function (see Example 2.2), and let $L_n = \{ w_n \}$. We remind the reader that Σ cannot be bounded by any recursive function (*cf.* Theorem 2.1).

The language L_n is accepted by a DTM $\mathcal{H}_n$ with $O(n)$ states and $O(1)$ working tape symbols, which simulates the n-state busy beaver $\mathcal{BB}_n$ and accepts an input $w \in a^*$ if and only if the space used by $\mathcal{BB}_n$ equals $|w|$. Since the simulation can be aborted if $\mathcal{BB}_n$ tries to use more than $|w|$ space, the machine $\mathcal{H}_n$ is bounded. Furthermore, the simulation of $\mathcal{BB}_n$ does not depend on inputs long enough, since it is interrupted otherwise. Hence, it is made in constant time. This allows to conclude that, with respect to the input length, $\mathcal{H}_n$ works in linear time and, so, it is a Hennie machine. Furthermore, any 1DFA would require $\Sigma(n)$ states to accept L_n. This completes the proof. □

7.2 Weight-Reducing Machines: Decidability, Expressiveness and Descriptional Complexity

In Section 7.1 we proved that it cannot be decided whether a bounded DTM works in linear time. Here, we show that this property becomes decidable for weight-reducing machines (even without requiring that they are bounded). Furthermore, each linear-time DTM $\mathcal{T}$ can be transformed into an equivalent weight-reducing machine whose size is bounded by a function of the size and of the running time of $\mathcal{T}$.

We also present a simulation of weight-reducing machines by finite automata, thus concluding that weight-reducing machines express exactly the class of regular languages. From such a simulation we shall obtain the size trade-off between weight-reducing machines and finite automata which, hence, is recursive. This contrasts with the non-recursive trade-off from Hennie machines to finite automata, proved in Section 7.1.

Proposition 7.1. *There exists an algorithm that decides in linear time whether a* DTM *is weight-reducing or not.*

Proof. Let $\mathcal{T} = \langle Q, \Sigma, \Gamma, \delta, q_0, F \rangle$ be a DTM. To decide if there is an order $<$ on Γ proving that $\mathcal{T}$ is weight-reducing, it suffices to check if the directed graph $G = \langle \Gamma, E \rangle$, with $E = \{ (\tau, \sigma) \mid \exists p, q \in Q \; \exists d \in \{ -1, 0, +1 \} : \delta(p, \sigma) = (q, \tau, d) \}$, is acyclic (each topological ordering of G acts as the required order $<$). All this is done in $O(\#\Gamma + \#E)$ time, *i.e.*, in time linear in the size of $\mathcal{T}$. □

We now study how linear-time DTMs can be made weight-reducing. To this aim, we use the fact that each DTM working in linear time makes a constant number of visits to each tape cell, hence linear-time is equivalent to a constant number of visits per tape cell. This property is stated in the following lemma, which derives from [Hen65, Proof of Theorem 3].

Lemma 7.1. *If a* DTM $\mathcal{T} = \langle Q, \Sigma, \Gamma, \delta, q_0, F \rangle$ *has time complexity* $t(n) \leq Kn$*, where* K *is a constant, then the number of instructions performed by* $\mathcal{T}$ *on any tape cell is at most* $2K \cdot (\#Q)^K + K$.

The following lemma, which will be used in this section to study trade-offs between the computational models we are investigating and finite automata, presents a transformation from linear-time Turing and Hennie machines into equivalent weight-reducing ones.

Lemma 7.2. *Let* $\mathcal{T} = \langle Q, \Sigma, \Gamma, \delta, q_0, F \rangle$ *be a linear-time* DTM *such that, for any input,* $\mathcal{T}$ *performs at most* k *computation steps on each tape cell, for some* $k > 0$*. Then there is a* wr DTM $\mathcal{A}$

accepting $L(\mathcal{T})$ with the same set of states Q as $\mathcal{T}$ and working alphabet of size $O(k \cdot \#\Gamma)$. Furthermore, if $\mathcal{T}$ is a Hennie machine then $\mathcal{A}$ is a weight-reducing Hennie machine.

Proof. To obtain $\mathcal{A}$ we incorporate a counter into the working alphabet of $\mathcal{T}$. For each scanned cell, the counter says what is the maximum number of visits $\mathcal{A}$ can perform during the remaining computation steps over the cell. Formally, define $\mathcal{A} = \langle Q, \Sigma, \Gamma', \delta', q_0, F \rangle$ with $\Gamma' = \Sigma \cup (\Gamma \times \{1, \ldots, k\})$ and, for all $q, q' \in Q$, $a, a' \in \Gamma$, $d \in \{-1, +1\}$ where $\delta(q, a) = (q', a', d)$, δ' fulfills

$$\begin{aligned} \delta'(q, a) &= (q', (a', k), d), \\ \delta'(q, (a, i)) &= (q', (a', i-1), d), \quad \text{for all } i \in \{2, \ldots, k\}. \end{aligned}$$

Using an ordering $<$ on Γ' such that

$$\begin{aligned} (a, i) &< b && \text{for all } a, b \in \Sigma, 1 \leq i \leq k, \text{ and} \\ (a, i) &< (b, j) && \text{for all } a, b \in \Sigma, 1 \leq i < j \leq k, \end{aligned}$$

it is easy to see that $\mathcal{A}$ is a wrDTM equivalent to $\mathcal{T}$. Furthermore if $\mathcal{T}$ is bounded, then $\mathcal{A}$ is also bounded and, hence, it is a wrDHM. □

We now investigate the transformation of weight-reducing machines into equivalent finite automata and its cost. To do that, let us start with some preliminary observations.

Let us fix a wrDTM $\mathcal{T} = \langle Q, \Sigma, \Gamma, \delta, q_0, F \rangle$. Assume that $\mathcal{T}$ never re-enters the initial state q_0 and that it can reach a final state in F only when scanning the rightmost cell which is visited during the computation. Any wrDTM can be modified to fulfill these restrictions by adding a constant number of states and working symbols.

Let $\rho = (\mathcal{C}_0, \mathcal{C}_1, \ldots)$ be the computation of $\mathcal{T}$ over a non-empty input $w \in \Sigma^+$. Let $q(j)$ denote the state of $\mathcal{T}$ in configuration $\mathcal{C}_j$. Similarly, let $a(j)$ denote the symbol scanned by $\mathcal{T}$ in $\mathcal{C}_j$.

Let τ_1, τ_2 and τ_3 denote the portion of $\mathcal{T}$'s tape which stores the initial blank symbols preceding w, the input w and the initial blank symbols following w, respectively.

It is possible to obtain, for each cell C of the tape of $\mathcal{T}$, the time-ordered sequence of states entered while visiting C. Notice that this kind of sequences are a variant of crossing sequences: they are defined by considering the states reached while the head is on the cell instead of considering the states while the head crosses one of the borders of C.[2]

It is possible to observe that ρ is an accepting computation if and only if the initial state q_0 occurs only once in the sequences, more precisely it is the first element of the sequence associated with the leftmost cell of τ_2, initially containing the first symbol of w, and a final state is the last element of the sequence associated with the rightmost visited cell.

Let $\mathcal{C}_{j_1}, \ldots, \mathcal{C}_{j_k}$, where $j_1 < \cdots < j_k$, be the sequence of all configurations in which $\mathcal{T}$ scans a tape cell C of τ_2 or τ_3. Observe that $a(j_1)$ and $q(j_1), \ldots, q(j_k)$ determine $a(j_i)$ for all $i = 2, \ldots, k$. For each configuration $\mathcal{C}_{j_i}$, it is also clear from which directions the head entered C and in which direction it moves out of it (C is always entered from the left neighboring cell the first time it is visited, *i.e.*, in configuration $\mathcal{C}_{j_1}$, with the only exception of the leftmost cell of τ_2, from which the computation starts with configuration $\mathcal{C}_1$, being in the initial state q_0; for $i > 1$, the cell C is entered in configuration $\mathcal{C}_{j_i}$ from the same direction it was left from configuration $\mathcal{C}_{j_{i-1}}$). For this reason, we can determine for two neighboring cells C_1, C_2 of τ_2 and τ_3 whether two sequences of states assigned with them are *consistent with the transitions of* $\mathcal{T}$, in the sense that each rightward head movement outgoing C_1 is an incoming leftward movement to C_2 (or, in other words, if, according to δ, $\mathcal{T}$ performs a rightward transition from a state in the sequence associated with C_1 to a state in the sequence associated with C_2) and each leftward head movement outgoing C_2 is an incoming rightward movement to C_1.

Similarly, we can determine whether a sequence of states assigned to the first cell of τ_2 is consistent with the transitions of $\mathcal{T}$ performed over the cell and over τ_1.

Hence, by summarizing, the following lemma can be proved.

Lemma 7.3. *Let $\mathcal{T} = \langle Q, \Sigma, \Gamma, \delta, q_0, F \rangle$ be a* wr DTM. *$\mathcal{T}$ accepts an input word w if and only if it*

[2]This kind of sequences are also known in the literature as *"slices of computations"* [CL15].

is possible to associate, with each cell C of the tape, a time-ordered sequence of states entered while visiting C consistent with the transition function δ of $\mathcal{T}$.

This lemma is used for the simulation of weight-reducing machines by finite automata that we now present.

Theorem 7.3. *For every wr* DTM $\mathcal{T} = \langle Q, \Sigma, \Gamma, \delta, q_0, F \rangle$ *there exist a* 1NFA *and a* 1DFA *accepting* $L(\mathcal{T})$ *with* $2^{O(\#\Gamma \cdot \log(\#Q))}$ *and* $2^{2^{O(\#\Gamma \cdot \log(\#Q))}}$ *states, respectively.*

Proof. We first describe a 1NFA $\mathcal{A} = \langle Q', \Sigma, \delta', q_-, F' \rangle$ which accepts $L(\mathcal{T})$, working on the principle of guessing the time-ordered sequences of states in which $\mathcal{T}$ scans each of the tape cells storing the input, and the symbol to be scanned next.

The weight-reducing property guarantees that $\mathcal{T}$ scans a cell C in at most $\#\Gamma + 1$ configurations. The set of states Q' thus consists of a special initial state q_- and all sequences of the form $(a, q_1, \ldots, q_k)$ where $a \in \Sigma \cup \{ \not\emptyset \}$, $1 \leq k \leq \#\Gamma + 1$, and $q_i \in Q$ for all $i \in \{ 1, \ldots, k \}$.

The set of final states F' consists of those states $(a, q_1, \ldots, q_k) \in Q'$ where $k > 0$, $q_k \in F$ and $a = \not\emptyset$. In addition, if $\varepsilon \in L(\mathcal{T})$, then F' also contains the initial state q_-.

Based on the observations leading to Lemma 7.3, transitions of $\mathcal{A}$ are defined by the following rules, which apply to any $a, b \in \Sigma$ and $p_i, q_i \in Q$.

- $(a, p_1, \ldots, p_\ell) \in \delta'(q_-, \varepsilon)$, where $p_1 = q_0$, if and only if the time-ordered sequence of states $(p_1, \ldots, p_\ell)$ is the one associated with the first cell of τ_2, initially storing a.
- $(b, p_1, \ldots, p_\ell) \in \delta'((a, q_1, \ldots, q_k), a)$ if and only if $(p_1, \ldots, p_\ell)$ and $(q_1, \ldots, q_k)$ are consistent with the transitions of $\mathcal{T}$ over two neighboring cells of τ_2 initially storing a and b, respectively.
- $(\not\emptyset, p_1, \ldots, p_\ell) \in \delta'((a, q_1, \ldots, q_k), a)$ if and only if $(p_1, \ldots, p_\ell)$ and $(q_1, \ldots, q_k)$ are consistent with the transitions of $\mathcal{T}$ over the last cell of τ_2, initially storing a, and the first cell of τ_3.

- $(\not\vdash, p_1, \ldots, p_\ell) \in \delta'((\not\vdash, q_1, \ldots, q_k), \varepsilon)$ if and only if $(p_1, \ldots, p_\ell)$ and $(q_1, \ldots, q_k)$ are consistent with the transitions of $\mathcal{T}$ over two neighboring cells of τ_3.

According to Lemma 7.3, the defined transitions ensure that $\mathcal{A}$ reaches an accepting configuration if and only if $\mathcal{T}$ accepts w.

The number of states of $\mathcal{A}$ is $1 + (\#\Sigma + 1)\sum_{i=0}^{\#\Gamma+1} n^i = 2^{O(\#\Gamma \log \#Q)}$. If $\mathcal{A}$ is in turn transformed to an equivalent 1DFA, the resulting automaton has $2^{2^{O(\#\Gamma \log \#Q)}}$ states. □

As a direct consequence of Theorem 7.3, we get that wrDTMs recognize exactly the class of regular languages.

Corollary 7.1. *A language is regular if and only if it is accepted by some* wr DTM.

Theorem 7.3 gives a double exponential upper bound for the size cost of the simulation of wrDTMs by 1DFAs. We now prove a matching lower bound.

To this end, for each $n \in \mathbb{N}$, we consider the family of languages $(B_n)_{n=0}^{\infty}$, in which, for each $n \in \mathbb{N}$, $B_n \subseteq \{0, 1, \$\}^*$ consists of strings $v_1\$v_2\$\cdots\$v_j$ where $j \in \mathbb{N}$, every $v_i \in \{0, 1\}^*$, $|v_j| \leq n$ and there is $\ell < j$ such that $v_\ell = v_j$. Informally, every string in B_n is a sequence of binary substrings which are separated by the symbol \$. Moreover, the last substring is of length at most n and it is a copy of one of the preceding substrings. For example,

$$v_1\$v_2\$v_3\$v_4\$v_5\$v_6 = 11\$0101110\$011\$0011\$001\$011 \in B_4$$

since $v_3 = v_6$ and $|v_6| \leq 4$.

Lemma 7.4. *For every $n \in \mathbb{N}$, the language B_n is accepted by a* wr DHM *with $O(1)$ states and $O(n)$ working symbols.*

Proof. Let $\Sigma = \{0, 1, \$\}$. We first describe a bounded DTM $\mathcal{T}$ accepting B_n, then we transform it into a wrDHM with the desired properties. Define the working alphabet of $\mathcal{T}$ as $\Gamma = \{0, 1, \$, x, f, \not\vdash\}$.

Let $w \in \Sigma^*$ be an input string of the form $w = v_1\$v_2\$\cdots\$v_k$ where each $v_i \in \{0, 1\}^*$, and $v_k = a_1 \cdots a_\ell$ where $a_i \in \{0, 1\}$ for all $i = 1, \ldots, \ell$. The machine $\mathcal{T}$ performs

$\min\{\ell, n\}$ iterations. In each iteration $\mathcal{T}$ moves the head from the leftmost position to the right end of w and back. It also rewrites some of the tape cells during this movement. Within the first iteration it stores a_ℓ in its finite control, and overwrites it by symbol x. Then it moves the head leftwards. Whenever it passes the symbol \$ and enters the right end of a substring v_j, it checks if its last symbol equals a_ℓ. If yes, it overwrite it with x, otherwise it overwrites it with f. During the i-th iteration, $\mathcal{T}$ finds $a_{\ell+1-i}$ (it is in the rightmost input cell not storing the symbol x), memorizes it in the finite control, overwrites the cell contents with x and checks the i-th symbol from behind of each v_j, with $j < k$, whether it matches $a_{\ell+1-i}$ (if yes, it overwrites the symbol with x, if not it rewrites it by f).

The initial contents of the tape and the outcomes of all iterations are illustrated by the following example:

```
11$0101110$011$0011$001$011
1x$010111f$01x$001x$00x$01x
xx$01011xf$0xx$00xx$0fx$0xx
xx$0101fxf$xxx$0xxx$xfx$xxx
```

$\mathcal{T}$ accepts w during the last iteration if and only if all symbols of v_k have been rewritten to x (this ensures $|v_k| \leq n$) and there is some v_j with all symbols also rewritten to x. In the above example, $\mathcal{T}$ accepts the input since all three symbols in v_3 have been rewritten to x.

The described bounded DTM $\mathcal{T}$ can be implemented as a wrDHM $\mathcal{H}$ with the working alphabet $\Gamma' = \Gamma \cup (\Gamma \times \{1, \ldots, 2n\})$ and a constant number of states. The working alphabet is constructed in the same way as in the proof of Lemma 7.2. It allows to perform $2n$ transitions over each tape cell (*i.e.*, 2 transitions for each iteration of $\mathcal{T}$). If $|v_k| > n$, it will happen that $\mathcal{H}$ attempts to start the $n+1$-st iteration, but this results in halting in a rejecting configuration. Hence, $\mathcal{H}$ is a wrDHM proving the lemma. □

Lemma 7.5. *Each* 1DFA *accepting* B_n *has at least* 2^{2^n} *states.*

Proof. Let $\mathcal{S}$ be the family of all subsets of $\{0,1\}^n$. Consider a subset $S = \{ w_1, \ldots, w_k \} \in \mathcal{S}$, where $w_1 < \cdots < w_k$ in the lexicographical order. Let us represent the subset S by the string $w(S) = w_1\$w_2\$\cdots\$w_k$, where the strings w_i are separated by the symbol \$. Let S_1 and S_2 be two different elements of $\mathcal{S}$ and let $u \in \{ 0,1 \}^n$ be a string which is in S_1 but not in S_2 (or vice versa). Then, $w(S_1)\$u \in B_n$ and $w(S_2)\$u \notin B_n$ (or vice versa), hence $\$u$ is a distinguishing extension, and, by the Myhill-Nerode theorem [Ner58], each 1DFA accepting B_n has at least $\#\mathcal{S} = 2^{2^n}$ states. □

From Theorem 7.3 and Lemmas 7.4 and 7.5, we obtain:

Corollary 7.2. *The size trade-offs from* wr DTMs *and* wr DHMs *to* 1DFAs *are double exponential.*

As shown in Theorem 7.2, by dropping the weight-reducing assumption for machines, the size trade-offs in Corollary 7.2 become not recursive. However, by taking into account also time complexity, we obtain the following result:

Theorem 7.4. *For every* DTM $\mathcal{T} = \langle Q, \Sigma, \Gamma, \delta, q_0, F \rangle$ *with time complexity* $t(n) \leq Kn$*, where* K *is a constant, there exist a* 1NFA *and a* 1DFA *accepting* $L(\mathcal{T})$ *with* $2^{O(k \cdot \#\Gamma \cdot \log(\#Q))}$ *and* $2^{2^{O(k \cdot \#\Gamma \cdot \log(\#Q))}}$ *states, respectively, where* $k = K \cdot (\#Q)^K + K$*.*

Proof. Consequence of Lemmas 7.1 and 7.2 and Theorem 7.3. □

7.3 Weight-Reducing Machines: Space and Time Usage, Haltingness

Weight-reducing Turing machines generalize weight-reducing bounded Turing machines (that are necessarily Hennie machines) by allowing to use additional tape cells that were not initially containing the input and to which we refer as *initially-blank cells*. This extension allows in particular infinite computations. For instance, a wrDTM can perform forward moves forever, rewriting each blank cell with some symbol. We now show that, however, due to the weight-reducing property, the amount of initially-blank cells that is

really useful, *i.e.*, that is visited in some halting computation, is bounded by some constant which can be computed from the size of the wrDTM and does not depend on the input string. This allows us to transform any wrDTM into an equivalent halting one of polynomial size, which therefore operates in linear time. Notice that Theorem 7.3 already gave a simulation of wrDTMs by a halting and linear-time computational model.

Lemma 7.6. *Let $\mathcal{T}$ be an n-state wr DTM which uses g working symbols. A computation of $\mathcal{T}$ is infinite if and only if it visits $(n+1)^g$ consecutive initially-blank cells,* i.e., *tape cells to the left or to the right of the initial segment.*

Proof. Since $\mathcal{T}$ is weight reducing, the number of visits to each tape cell is bounded by a constant which, in turn, is bounded by g. Thus, each infinite computation should visit infinitely many tape cells, hence at least $(n+1)^g$ consecutive initially-blank cells.

To prove the converse, let us consider a halting computation ρ of $\mathcal{T}$ over an input word of length ℓ. Let $\nu : \mathbb{Z} \to Q^*$ be the function which maps each cell position to the sequence of states entered at this position in ρ. Because $\mathcal{T}$ is weight-reducing, $|\nu(i)| \leq g$ for each position i. If there exist two positions i and j, laying at the same side of the initial segment, *i.e.*, $j < i \leq 0$ or $\ell < i < j$, such that $\nu(i) = \nu(j)$, then, by determinism of $\mathcal{T}$, $\nu(j + k(j-i)) = \nu(i)$ for each $k > 0$. This yields an infinite amount of positions with the same associated sequence of states by ν. By haltingness of ρ, these repeated sequences are necessarily empty. Thus, no two such positions on the same side of the initial segment are mapped by ν to the same nonempty sequence. Since there are less than $(n+1)^g$ nonempty distinct sequences of states of length at most g, we conclude that the number of consecutive initially-blank cells visited during the computation is less than $(n+1)^g$. □

Proposition 7.2. *By a polynomial size increase, each wr DTM can be transformed into an equivalent halting linear-time one.*

Proof. From an n-state wrDTM $\mathcal{T}$ with a working alphabet of cardinality g, we can build an equivalent halting wrDTM $\mathcal{T}'$ which works as follows. First $\mathcal{T}'$ marks $(n+1)^g$ initially-blank cells to the left and to the right of the initial segment. This is obtained by using

a counter in basis $(n+1)$, stored on g consecutive tape cells, which is incremented up to $(n+1)^g$ and shifted along the tape. Once the space is marked, $\mathcal{T}'$ simulates $\mathcal{T}$, stopping and rejecting if the simulation reaches a blank cell.

From Lemma 7.6, we can easily conclude that each halting computation of $\mathcal{T}$ is simulated by an equivalent halting computation of $\mathcal{T}'$, while each infinite computation of $\mathcal{T}$ is replaced in $\mathcal{T}'$ by a computation which reaches a blank cell and then stops and rejects. It can be verified that the size of $\mathcal{T}'$ is polynomial in n and g. □

Using Lemma 7.6 and Proposition 7.2, we prove the following property.

Theorem 7.5. *It is decidable whether a* wr DTM *halts on each input string.*

Proof. From any given wrDTM $\mathcal{T}$, we construct a halting wrDTM $\mathcal{T}'$ which, besides all the strings accepted by $\mathcal{T}$, accepts all the strings on which $\mathcal{T}$ does not halt. To this aim, we can slightly modify the construction used to prove Proposition 7.2, in such a way that when the head reaches a blank cell outside the initial segment and the initially marked space, the machine stops and accepts. Hence, the given wrDTM $\mathcal{T}$ halts on each input string if and only if the finite automata which are obtained from $\mathcal{T}$ and $\mathcal{T}'$ according to Theorem 7.3 are equivalent. □

As a consequence:

Corollary 7.3. *It is decidable whether a* wr DTM *works in linear time.*

As seen in Lemma 7.6, the space used by a halting wrDTM is $m + C$ where m is the input length and C is a constant. By contrast, according to the definition, the space used by a wrDHM is m. Over long enough inputs, the constant C can be eliminated. Thus, every wrDTM can be simulated by a wrDHM of polynomial size on long inputs. This result will be useful in next sections.

Lemma 7.7. *Let $\mathcal{T}$ be a weight-reducing Turing machine which uses at most C initially-blank cells in every halting computation. Then, there exists a weight-reducing Hennie machine $\mathcal{H}$ of size polynomial in the size of $\mathcal{T}$, which agrees with $\mathcal{T}$ on every input of length at least C.*

Proof. When the input is long enough, the tape parts containing useful initially-blank cells can be folded as a second track over the initial segment, paying a polynomial increase of the size of the machine. □

7.4 Simulating Two-Way Automata by Weight-Reducing Machines

This section is devoted to present our main simulation: we show that every n-state 2NFA $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F \rangle$ can be transformed into an equivalent wrDTM of size polynomial in the size of $\mathcal{A}$. Our construction is based on the classical simulation of 2NFAs by 1DFAs, inspired from Shepherdson's construction [She59]. The main idea is to perform forward moves, while updating a table of size n^2 that describes parts of computations which may occur to the left of the current position, analogously to the technique used for the linear-time simulation of 1-LAs. In parallel, an adaptation of the powerset construction is done, in such a way that the set of states that are accessible from the initial configuration when visiting for the first time the current head position is updated at each move. In the simulation by 1DFAs, the table and the set are stored on the finite state control. In our simulation by wrDTMs they will be written, under a suitable encoding, in $O(n^2)$ many tape cells.

The construction is similar to the one for the linear-time simulation of 1-LAs presented in Chapter 6. More precisely, we shall use a slight modification of the technique shown in Section 6.2.1.2 to describe computations paths occurring on some restricted part of the tape. In particular, for a prefix $z'X$ of the tape content, we build the set $\tau_{z'X}$, composed by the pairs of states (p, q) such that there exists a computation path starting from p with the head scanning the last symbol of $z'X$ and ending one cell to the right of such a symbol in the state q. Additionally, we define the set $\gamma_{z'X}$ of states that are reachable *from the initial configuration*, when visiting for the first time the position to the right of the part containing $z'X$. Formally, for $z'X \in \{ \rhd \} \cdot \Sigma^* \cdot \{ \varepsilon, \lhd \}$ a prefix of the tape content

with $|X| = 1$:

$$\tau_{zX} = \{\, (p,q) \in Q \times Q \mid\ zpX \vdash^* zXq \,\}, \text{and}$$
$$\gamma_{zX} = \{\, r \in Q \mid\ q_0 zX \vdash^* zXr \,\}.$$

Observe that a word $w \in \Sigma^*$ is accepted by $\mathcal{A}$ if and only if $F \cap \gamma_{\triangleright w \triangleleft} \neq \emptyset$. In order to simulate $\mathcal{A}$ on input w, it is thus sufficient to incrementally compute γ_z for each prefix z of $\triangleright w \triangleleft$. To do so, we will keep updated the table τ_z as well. Indeed, given γ_z, τ_z and a symbol σ, it is possible to compute $\gamma_{z\sigma}$ and $\tau_{z\sigma}$. In particular, the set $\tau_{z\sigma}$ is obtained from τ_z as explained in Section 6.2.1.2 (see Figure 6.2), while it is possible to observe that $q \in \gamma_{z\sigma}$ if and only if there exists $p \in \gamma_z$ such that $(p,q) \in \tau_{z\sigma}$.

We represent a pair (γ_z, τ_z) as a word uv in $\{\,0,1\,\}^*$ with $|u| = n$ and $|v| = n^2$. Each bit of u (resp., v) indicates the membership of some state p (resp., some pair (p,q) of states) to the set γ_z (resp., τ_z) through an implicitly fixed bijection from Q to $\{\,1,\ldots,n\,\}$ (resp., from Q^2 to $\{\,1,\ldots,n^2\,\}$). From such a word uv, and given a fixed input symbol σ, a wrDHM is able to compute the representation $u'v'$ of the pair $(\gamma_{z\sigma}, \tau_{z\sigma})$. Notice that such computations do not depend on z, *i.e.*, z is not given to the machine.

Lemma 7.8. *For each $\sigma \in \Sigma$, there exists a wrDHM $\mathcal{T}_\sigma$ of size polynomial in n that on input (γ_z, τ_z) halts with the tape containing $(\gamma_{z\sigma}, \tau_{z\sigma})$. The input and the output are represented on the tape as strings in $\{\,0,1\,\}^{n+n^2}$.*

Proof. Fixed σ, let uv be the input string encoding the pair of tables (γ_z, τ_z), of size n and n^2 respectively. In order to update them, $\mathcal{T}_\sigma$ uses a second track on the tape, on which it will progressively build the updated tables. At the end of the computation, namely when the updated tables have been determined and written down over the second track, the device performs a projection of the tape on its second track, in order to produce the correct output, and halts.

We fix the working alphabet $\Gamma = \{\,0,1\,\} \cup \{\,0,1\,\}^2$. The "simple" symbols from $\{\,0,1\,\}$ are used only for the input and the output of $\mathcal{T}_\sigma$. For now on, we suppose that the tape contains only symbols from the 2-track alphabet part, *i.e.*, the right side of the union.

Moreover, since the length of the input is $n + n^2$, we can suppose that $\mathcal{T}_\sigma$ keeps updated a state component of size $n + n^2$ which always stores the exact position of its head on the tape. This allows it to navigate over the tables.

We divide the tape into two parts: a prefix $\overline{u}$ of length n (thus covering the factor u which encodes γ_z on its first track) and a suffix $\overline{v}$ of length n^2 (thus covering the factor v which encodes τ_z on its first track). As previously explained, the updated table $\gamma_{z\sigma}$ can easily be obtained once the updated table $\tau_{z\sigma}$ has been computed. Hence, we first show how to write the table $\tau_{z\sigma}$ on the second track of $\overline{v}$. This is achieved using the space n available on the second track of $\overline{u}$ as temporary memory, to which we refer as *temporary table*.

It is possible to observe that (see Figure 6.2), a computation path on the segment containing $z\sigma$ starting from the rightmost position of the segment and exiting the segment to the right at its last step, *i.e.*, a computation of the form $zp\sigma \vdash^* z\sigma q$, can be decomposed into an alternation of computation paths on the segment containing z (described by the table τ_z) and of backward computation steps on σ connecting these paths, followed by a last forward computation step on σ that exits the segment. For each state p, in order to decide which pairs (p, q) belong to $\tau_{z\sigma}$, the machine $\mathcal{T}_\sigma$ first computes the set Z_p of states that are reachable at the rightmost position of the segment containing $z\sigma$, from the state p at the same position, by visiting only cells from the segment, *i.e.*,

$$Z_p = \{\, r \mid zp\sigma \vdash^* zr\sigma \,\}.$$

Thus, a pair (p, q) belongs to $\tau_{z\sigma}$ if and only if for some $r \in Z_p$, we have $(q, +1) \in \delta(r, \sigma)$. For a fixed p, $\mathcal{T}_\sigma$ can incrementally construct Z_p on the temporary table as follows. Initially, all the cells from the table are unmarked (*i.e.*, contain 0) except the one corresponding to state p which contains 1. The update process behaves as follows: for each state r corresponding to a marked cell, each state s such that $(s, -1) \in \delta(r, \sigma)$, and each state r' such that $(s, r') \in \tau_z$, the machine marks the cell corresponding to r' with 1 in the temporary table. Since Z_p has size bounded by n, after at most n passes, the temporary table is not modified anymore and contains exactly an encoding of Z_p.

So done, computing the set Z_p uses only a polynomial number of states in n. This is however not sufficient to get a weight-reducing machine of polynomial size. To this end we should indeed prove that the number of visits to each cell is bounded by some polynomial in n. To update Z_p, three nested loops on states, namely on r, s, and r', are used. Once such a triple is fixed, the machine navigates on the tape in order to check that r is currently marked in the temporary table, and $(s, r') \in \tau_z$ (notice that the condition $(s, -1) \in \delta(r, \sigma)$ is verified in constant time, since σ is fixed). These two conditions require to read the corresponding cells in the temporary table (on $\overline{u}$) and in the table τ_z (on $\overline{v}$), respectively. This can be performed by visiting each tape cell at most twice. Marking the cell corresponding to r' also implies to scan the tape part $\overline{u}$ twice. As the operation is repeated for each triple, we obtain that the number of visits to each cell in a pass for updating Z_p is in $O(n^3)$. Since the number of passes is at most n, the total number of visits to each cell is in $O(n^4)$.

Once Z_p has been computed, for each state r corresponding to a marked cell in the temporary table, and each state q such that $(q, +1) \in \delta(r, \sigma)$, $\mathcal{T}_\sigma$ adds the pair (p, q) to the table $\tau_{z\sigma}$ represented on the second track of $\overline{v}$. This requires to visit $O(n)$ times each tape cell and can be performed using only a polynomial number of states in n.

By repeating this for each state p, we manage to update the table from τ_z to $\tau_{z\sigma}$. Finally, we can update the table γ_z. As observed before, it is sufficient to consider for each state q, whether $(p, q) \in \tau_{z\sigma}$ for some $p \in \gamma_z$. This last step requires only polynomially many states and a linear number of visits to each cell. Combining the above-described subroutines, and taking into account the state component which stores the exact head position, we obtain a DHM with $O(n^6)$ states and $O(1)$ working symbols, whose number of visits to each tape cell is $O(n^5)$. The machine can finally be converted into an equivalent wrDHM of polynomial size (see Lemma 7.2). □

We are now ready to state our main simulation.

Theorem 7.6. *Every n-state* 2NFA *can be transformed into an equivalent halting* wr DTM *of size polynomial in n.*

Proof. Let $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F \rangle$ be a 2NFA. We build a deterministic Turing machine that mimics the simulation of $\mathcal{A}$ by a 1DFA: after reading any prefix z of an input w surrounded by endmarkers, the machine stores the tables γ_z and τ_z and finally checks the existence of a final state in $\gamma_{\triangleright w \triangleleft}$. The tables are stored on a suitable tape track and updated each time a further input symbol is read, using the method presented in Lemma 7.8. This can be achieved by switching between two tape tracks at each update of the tables. However, as the number of updates is linear in the length of the input, storing and updating the tables on a fixed part of length $n + n^2$ of the tape would lead to a non-weight-reducing Turing machine. To handle this issue, for each prefix z of $\triangleright w \triangleleft$, we store the tables γ_z and τ_z on the $n + n^2$ cells that precede the last position of z. (Remember that, in a wrDTM, some initially-blank cells to the left of the initial segment are available.) Thus, at each update of the tables made according to Lemma 7.8, the tables are shifted one cell to the right. Therefore, since a fixed cell may occur in $n + n^2$ successive stored tables, according to Lemma 7.8 the number of visits to this cell is bounded by some polynomial in n. We thus obtain a halting weight-reducing Turing machine equivalent to $\mathcal{A}$ whose size is polynomial in the size of $\mathcal{A}$. Furthermore, the machine uses only $n + n^2$ initially-blank cells, that are all to the left of the initial segment. □

7.5 Simulating Two-Way Automata by Hennie Machines

In Section 7.4 we provided a polynomial size conversion from 2NFAs to wrDTMs. The resulting machines use further tape cells, besides the initial segment. In this section we study how to make such a simulation when the use of such extra space is not allowed, namely when we want to obtain a deterministic Hennie machine. We show that a polynomial conversion still exists, but we are not able to guarantee that the resulting machine is weight reducing. Actually this issue is related to "short" inputs, namely to strings of length less than n^2, where n is the number of states of the given 2NFA. For such inputs we do not have enough tape space to perform the simulation in Theorem 7.6. We will deal with them, by using a different technique.

Let us start by considering acceptance of "long" inputs.

Theorem 7.7. *For each n-state* 2NFA $\mathcal{A}$*, there exists a* wr DHM $\mathcal{H}$ *of size polynomial in n which agrees with $\mathcal{A}$ on strings of length at least n^2.*

Proof. The technique used in the proof of Theorem 7.6 can be exploited, with slight modifications. Indeed, when recovering the tables corresponding to the "short" prefixes z's of the tape content, the wrDTM machine resulting from the above construction uses up to $n + n^2$ initially-blank cells to the left of the initial segment, that are not any longer available with a wrDHM. By folding n of these cells on an additional track, as in Lemma 7.7, we can reduce this space amount to n^2 cells. Moreover, when the input string is longer than n^2, the tape part containing these useful initially-blank cells can be folded as a further track over the initial segment. The size increase implied by both constructions is polynomial in n. □

In the case of unary 2NFAs, the number of short inputs that are not handled by Theorem 7.7 is n^2. They can be managed in a read-only preliminary phase which uses $O(n^2)$ states.

Corollary 7.4. *Every n-state unary* 2NFA *is equivalent to a* wr DHM *of size polynomial in n.*

In the nonunary case, since the number of short strings is exponential in n, we cannot apply the same technique as in Corollary 7.4. However, we are able to obtain a polynomial size Hennie machine, using a different technique, which is based on the analysis of the computation graph of the simulated 2NFA.

Theorem 7.8. *Each n-state* 2NFA *is equivalent to a* DHM *of size polynomial in n.*

Proof. Let $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F \rangle$ be a 2NFA, with $\#Q = n$. Without loss of generality we suppose $F = \{ q_f \}$. Let $w \in \Sigma^*$ be an input word, with $m = |w|$. We distinguish three cases, depending on m. Observe that the simulating DHM $\mathcal{H}$ can decide the case by performing a reading traversal of the input using a polynomial number of states.

If $m \geq n^2$, then $\mathcal{H}$ simulates $\mathcal{A}$ as in Theorem 7.7.

If $m \leq \log n$, then $\mathcal{H}$ simulates a 1DFA with a polynomial number of states in n, which agrees with $\mathcal{A}$ on all strings of length at most $\log n$.

Finally, if $\log n < m < n^2$, then $\mathcal{H}$ checks whether there is an accepting computation of $\mathcal{A}$ on w by analyzing the *computation graph* $G = \langle V, E \rangle$ of $\mathcal{A}$ on w whose nodes represent the configurations of $\mathcal{A}$ and whose edges represent single computation steps on input w. Since w is fixed, configurations can be represented as pairs $(q, i) \in V = Q \times \{0, \ldots, m+1\} \cup \{(q_f, m+2)\}$ indicating that $\mathcal{A}$ is in the state q while scanning the i-th symbol of the input tape. There exists an edge from (q, i) to (p, j) if and only if $(p, j-i) \in \delta(q, \widetilde{w}[i])$, where $\widetilde{w} = \triangleright w \triangleleft$.

The simulating Hennie machine $\mathcal{H}$ should check the existence of a computation of $\mathcal{A}$ starting from the initial state q_0 with the head on the left endmarker (*i.e.*, at position 0) and ending in the unique final state q_f after violating the right endmarker (*i.e.*, at position $m+2$). This is equivalent to check the existence of a path from the node $(q_0, 0)$ to the node $(q_f, m+2)$ in G. Since there are $n(m+2)+1$ nodes in G, if such a path exists, then there should exist one of length at most $K = n(m+2)$. Hence, checking the existence of an accepting computation reduces to checking the existence of a path of length at most K in G. The recursive function `reachable` is used to perform this checking by calling `reachable`$(q_0, 0, q_f, m+2, K)$.

Let us show how a call of `reachable`(p, i, q, j, T) works. The function has to check if there exists a computation from (p, i) to (q, j) of length at most T. This is done by using a *divide-and-conquer* technique as in the famous proof of the Savitch's Theorem [Sav70]. If $(p, i) = (q, j)$, *i.e.*, there is a path of length 0 from (p, i) to (q, j), then the function returns **true** independently of T (Line 94). Otherwise, if $T = 0$ but $(p, i) \neq (q, j)$, then the function returns **false** (Line 95), while, if $T = 1$, the function returns **true** if there is a suitable edge in G (Line 97). In order to verify that, $\mathcal{H}$ saves (q, i) and (p, j) in its internal state and then, if the distance between i and j is 1, it moves its head to the i-th position, reads the symbol $\widetilde{w}[i]$, and checks $(q, j-i) \in \delta(p, \widetilde{w}[i])$. Notice that this read-only process uses only a number of states polynomial in n.

Function 11: `reachable`(p, i, q, j, T): boolean

Checks the existence of a path from (p, i) to (q, j) of length less than or equal to T in the graph of the configurations of a given 2NFA on input $w = w_1 \cdots w_m$

```
94  if (p, i) = (q, j) then return true
95  if T = 0 then return false
96  if T = 1 then
97      if (q, j − i) ∈ δ(p, w̃[i]) then return true
98  else
99      foreach r, ℓ ∈ Q × { 0, ..., m + 1 } do
100         if reachable(p, i, r, ℓ, ⌊T/2⌋) then
101             if reachable(r, ℓ, q, j, ⌈T/2⌉) then return true
102 return false
```

In the recursive case, for checking if there exists a path in the graph from (p, i) to (q, j) of length at most $T > 1$, $\mathcal{H}$ verifies whether there exists a node $(r, \ell) \in Q \times \{ 0, \ldots, m + 1 \}$ such that there is a path from (p, i) to (r, ℓ) of length at most $\lfloor \frac{T}{2} \rfloor$ and a path from (r, ℓ) to (q, j) of length at most $\lceil \frac{T}{2} \rceil$ (Lines 99 to 101). This is done by trying all possible nodes (r, ℓ) until finding one satisfying these conditions. If it does not exist, then the procedure returns **false** (Line 102).

Recursive calls to the function `reachable` can be naturally saved on a pushdown store. More precisely, at the beginning of the simulation the store is empty. When a call to `reachable`(p, i, q, j, T) is performed, the activation record, consisting of the parameters p, i, q, j, and T, is pushed on the top of the pushdown. Similarly, when `reachable` returns, the activation record of the last call is popped off. The function `reachable` uses seven variables, five of them being arguments saved on the pushdown store, and two being the local variables r and ℓ. As these two local variables are arguments of inner recursive calls, their values can be recovered when popping off the inner activation record (after the corresponding call has returned). Hence, the state components saving r and ℓ are

freed at each recursive call. Therefore, all the checks performed by `reachable` can be done with a number of states that is polynomial in n, and using the pushdown alphabet $\left(Q \times \{\, 0, \ldots, n^2 \,\}\right)^2 \times \{\, 0, \ldots, \lceil \log K \rceil \,\}$ of size polynomial in n.

Finally, notice that the maximum recursion depth is $\lceil \log K \rceil = O(\log n)$. The stack of recursion calls can be stored in a separated track on $\log n$ tape cells, by using standard space compression techniques, that only induce a polynomial increase of the working alphabet. Since the input length m is larger than $\log n$, $\mathcal{H}$ has enough space in its initial segment. The number of visits to each cell is super-polynomial in n, hence the machine is not weight-reducing. However, because the input lengths are bounded by n^2, the number of visits to each cell is bounded by a number which only depends on n.

Considering also how the machine works on inputs of length at least n^2, this allows us to conclude that the working time of the whole machine $\mathcal{H}$ is linear in the input length.

□

It is natural to ask if Theorem 7.8 can be improved in order to obtain, from a given 2NFA $\mathcal{A}$, an equivalent `wr`DHM of polynomial size. In the light of Theorem 7.7, to do that it will be enough to obtain a `wr`DHM of polynomial size which agrees with the 2NFA on "short" inputs. With this respect, we point out that the problem of Sakoda and Sipser seems to be hard even when restricted to strings of length polynomial in the number of states of $\mathcal{A}$ [Kap14a].

7.6 Simulating One-Way Automata by Hennie Machines

In this section we restrict our attention to one-way automata simulations. A natural question is to ask if in the case of 1NFAs a result stronger than Theorem 7.8 can be achieved. A simulation of 1NFAs by `wr`DHMs was studied in [Prů14, Theorem 11], claiming that each n-state 1NFA $\mathcal{A}$ has an equivalent `wr`DHM $\mathcal{H}$ of size polynomial in n. Unfortunately, the presented proof is incorrect as it casts the problem of $\mathcal{A}$ acceptance as the problem of reachability in an undirected computation graph. Existence of a path connecting the

initial and an accepting configuration in such a graph does not guarantee the existence of an accepting computation of $\mathcal{A}$ since the path can include "back" edges.

By revising the result, we now show that each 1NFA can be simulated by a deterministic Hennie machine which, however, is not weight-reducing. The improvement with respect to Theorem 7.8 is that, in this case, short inputs are the strings of length less than n rather than n^2.

Let us start by presenting the weight-reducing simulation for "long" inputs.

Proposition 7.3. *For each n-state* 1NFA $\mathcal{A}$*, there exists a* wr DHM $\mathcal{H}$ *of size polynomial in n which agrees with $\mathcal{A}$ on strings of length at least n.*

Proof. Let $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F \rangle$ where $Q = \{ q_0, \ldots, q_{n-1} \}$. Let us assume the standard definition for 1NFAs. Hence, the computation of $\mathcal{A}$ starts in the initial state q_0 with the head scanning the first input symbol; at each step one symbol is read from the tape and the state is changed according to the transition function; the computation is accepting if a final state is reached after reading the last input symbol.

We first describe a wrDTM $\mathcal{T}$ simulating $\mathcal{A}$, which uses at most n further extra tape cells to the right of the initial tape segment. Then, according to Lemma 7.7, $\mathcal{T}$ will be converted into the wanted Hennie machine $\mathcal{H}$ which agrees with $\mathcal{A}$ on inputs of length at least n.

Let C_i denote the i-th tape cell of $\mathcal{T}$. Hence, on input $w \in \Sigma^*$ of length m, C_i initially contains the input symbol w_i, $i = 1, \ldots, m$. For each $i = 1, \ldots, \lceil \frac{m}{n} \rceil$, define a block B_i consisting of tape cells $C_{(i-1)n+1}, \ldots, C_{i \cdot n}$. Note that each block B_i can represent any subset $Q' \subseteq Q$ (mark the j-th cell of B_i if and only if $q_{j-1} \in Q'$). Let $Q_0 = \{ q_0 \}$ and, for $i = 1, \ldots, m$, let Q_i be the subset of states reachable by $\mathcal{A}$ after performing its i-th transition, i.e., after reading w_i.

The machine $\mathcal{T}$ initializes the simulation by representing Q_0 in B_1. For each i-th computation step of $\mathcal{A}$, the machine $\mathcal{T}$ computes Q_i based on Q_{i-1} and represents Q_i in block $B_{1+\lfloor \frac{i}{n} \rfloor}$. In the end, $\mathcal{T}$ checks if the representation of Q_m includes a state from F.

Since each block is used at most $2n + 2$ times to represent a subset of Q, the number

of transitions performed by $\mathcal{T}$ over each tape cell is upper bounded by a polynomial in n. This ensures that $\mathcal{T}$ is of size polynomial in n.

Finally, from $\mathcal{T}$ we obtain the wanted wrDHM $\mathcal{H}$ by applying Lemma 7.7, as already explained. □

Using a technique similar to that of Theorem 7.8, we prove the following.

Proposition 7.4. *Every n-state* 1NFA $\mathcal{A}$ *is equivalent to a* DHM $\mathcal{H}$ *of size polynomial in n.*

Proof. Let $w \in \Sigma^*$ be an input to $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F \rangle$ where, without loss of generality, $F = \{ q_f \}$. Distinguish two cases by $m = |w|$.

If $m \leq n$, the machine $\mathcal{H}$ checks whether there is an accepting computation of $\mathcal{A}$ for w by calling `reachable1`$(q_0, 0, q_f, m)$, a slightly modified version of the function `reachable` presented in the proof of Theorem 7.8, adapted to deal with 1NFAs.

Function 12: `reachable1`(p, i, q, j): boolean
Checks the existence of a path from (p, i) to (q, j) of length $j - i$ in the graph of the configurations of a given 1NFA on input $w = w_1 \cdots w_m$

```
103 if (p, i) = (q, j) then return true
104 if j − i = 1 and q ∈ δ(p, w_j) then return true
105 if j − i > 1 then
106     foreach r ∈ Q do
107         if reachable1(p, i, r, ⌊(i + j)/2⌋) then
108             if reachable1(r, ⌈(i + j)/2⌉, q, j) then return true
109 return false
```

The recursion depth is $O(\log m)$. At each level of the recursion, the function needs to store states p, q, r and indices i, j. This can be done in a tape cell if $O(n^5)$ working tape symbols are provided for this purpose. The function runs in $n^{O(\log m)}$ time.

If $m > n$, the Hennie machine $\mathcal{H}$ executes the computation described in the proof of Theorem 7.3.

In conclusion, the constructed machine $\mathcal{H}$ has the number of states and working tape symbols polynomial in n, hence it is of size polynomial in n. Note that the number of transitions performed by $\mathcal{H}$ over any tape field is $n^{O(\log n)}$. □

By summarizing, from Proposition 7.3 and Proposition 7.4 we obtain:

Corollary 7.5. *For each n-state* 1NFA $\mathcal{A}$*, there exist:*

- *a* wr DHM *of size polynomial in n which agrees with* $\mathcal{A}$ *on words longer than n;*
- *an equivalent* DHM *of size polynomial in n.*

8

Pushdown Automata and Space Restrictions

As discussed in Chapter 4, pushdown automata in which the maximum height of the pushdown is limited by some constant, namely constant-height pushdown automata, allow more succinct representations of regular languages than finite automata [GMP10], and are polynomially related in size with their natural generative counterpart, non-self-embedding context-free grammars, roughly, context-free grammars without "true" recursion [Cho59a].

In this chapter we turn our attention on standard pushdown automata, namely with an unrestricted pushdown store, that, however, are able to accept their inputs by making use only of a constant amount of the pushdown store. More precisely, we say that a pushdown automaton $\mathcal{M}$ *accepts in constant height* h, for some given h, if for each word in the language accepted by $\mathcal{M}$ there exists one accepting computation in which the maximum height reached by the store is bounded by h. Notice that this does not prevent the existence of accepting or rejecting computations using an unbounded pushdown height.

It is a simple observation that a pushdown automaton $\mathcal{M}$ accepting in constant height h can be converted into an equivalent constant-height pushdown automaton: in any configuration it is enough to keep track of the current height in order to stop and reject when a computation tries to exceed the height limit. The description of the resulting constant-height pushdown automaton has size polynomial in h and in the size of the description of $\mathcal{M}$.

While studying these size relationships, we tried to understand *how large can h be with respect to the size of the description of* $\mathcal{M}$. We discovered that h can be arbitrarily large. Indeed, in the first part of the chapter we prove that there is no recursive function bounding the maximal height reached by the pushdown store in a pushdown automaton accepting in constant height, with respect to the size of its description.

We also prove that it cannot be decided if a pushdown automaton accepts in constant height.

In the second part of the chapter we restrict the attention to the case of unary pushdown automata. By studying the structure of the computations of these devices, we are able to prove that, in contrast to the general case, it can be decided whether or not they accept in constant height. Furthermore, we also prove that if a unary pushdown automaton $\mathcal{M}$ accepts in height h, constant with respect to the input length, then h is bounded by an exponential function in the size of $\mathcal{M}$. By presenting a suitable family of pushdown automata, we show that this bound cannot be reduced.

In the final part of the chapter we consider pushdown automata that accept using height which is not constant in the input length. Our aim is to investigate how the pushdown height grows. In particular, we want to know if there exists a minimum growth of the pushdown height, with respect to the length of the input, when it is not constant. The answer to this question is already known and it derives from results on Turing machines: the height of the store should grow at least as a double logarithmic function [Alb85]. This lower bound cannot be increased, because a matching upper bound has been recently obtained in [Bed+16]. As a consequence of the constructions presented in the second part of the chapter, we are able to prove that in the unary case this lower bound is logarithmic. We also show that it cannot be further increased.

The results discussed in this chapter have been presented in [PP19b].

8.1 Preliminary Definitions and Results

Let us start by presenting the main measure we consider in the chapter, namely the *pushdown height*. The height of a PDA $\mathcal{M}$ in a given configuration is the number of symbols in the pushdown store *besides* the start symbol Z_0. Hence, in the initial and in the accepting configurations the height is 0. The height in a computation $\mathcal{C}$ is the maximum height reached in the configurations occurring in $\mathcal{C}$.

We say that $\mathcal{M}$ uses height $h(x)$ on an accepted input $x \in \Sigma^*$ if and only if $h(x)$ is the minimum pushdown height necessary to accept such a string, namely, there exists a computation accepting x using pushdown height $h(x)$, and no computations accepting x using height smaller than $h(x)$. Moreover, if x is rejected then $h(x) = 0$. To study pushdown height with respect to input lengths, we consider the worst case among all possible inputs of the same length. Hence, we define $h(n) = \max\{\, h(x) \mid x \in \Sigma^*, |x| = n \,\}$. When there is a constant H such that, for each n, $h(n)$ is bounded by H, we say that $\mathcal{M}$ *accepts in constant height*. Each PDA accepting in constant height can be easily transformed into an equivalent finite automaton. So the language accepted by it is regular.

We now present some technical notions and results that will be useful in order to state our results. Let $\mathcal{M} = \langle Q, \Sigma, \Gamma, \delta, q_I, Z_0, \{\, q_F \,\}\rangle$ be a fixed PDA.

A *surface pair* is defined by a state $q \in Q$ and a symbol $A \in \Gamma$, and it is denoted by $[qA]$. The surface pair in a given configuration is defined by the current state and the topmost stack symbol, namely the only part of the stack which is relevant in order to decide the next move.

A *surface triple* is defined by two states $q, p \in Q$ and a symbol $A \in \Gamma$, and it is denoted by $[qAp]$. Surface triples are used to study parts of computations starting and ending at the same pushdown height and that do not go below that height in between. More precisely, a $[qAp]$*-computation* on an input string $x \in \Sigma^*$ is a computation $\mathcal{C}$ which starts from the state q with A on the top of the pushdown at some height h and the input head on the tape cell containing the leftmost symbol of x, and ends in the state p with A on the top of the pushdown at the same height h and the input head on the cell to the right of

the cell containing the rightmost symbol of x, without reaching pushdown height smaller than h in between. We also say that $\mathcal{C}$ *consumes* the input x. We point out that, during $\mathcal{C}$, the symbol A at height h is never replaced. Hence, $\mathcal{C}$ does not depend on h and on the symbols in the pushdown store at height smaller than h. The *stack increment* during $\mathcal{C}$ is the difference between the maximum stack height in $\mathcal{C}$ and the stack height at the beginning and at the end of $\mathcal{C}$. Notice that the surface pairs at the beginning and at the end of $\mathcal{C}$ are $[qA]$ and $[pA]$, respectively.

We denote by $L_{[qAp]}$ the set of input strings consumed in all possible $[qAp]$-computations. We point out that the set of accepting computations of $\mathcal{M}$ coincides with the set of $[q_I Z_0 q_F]$-computations. Hence, $L_{[q_I Z_0 q_F]}$ is the language accepted by $\mathcal{M}$. Furthermore, for each surface triple $[qAp]$, by suitably modifying $\mathcal{M}$, we can obtain a PDA accepting $L_{[qAp]}$ which, hence, is context free.

A *horizontal loop* on a surface pair $[qA]$ is any $[qAq]$-computation consuming *at least one input symbol*. By considering a computation of 0 moves, we always have $\varepsilon \in L_{[qAq]}$. Hence $[qA]$ *has a horizontal loop* when $L_{[qAq]}$ contains at least one more string, besides ε. Using standard arguments on context-free languages, the following result can be proved:

Lemma 8.1. *It is decidable if a surface pair $[qA]$ has a horizontal loop.*

Proof sketch. By using standard transformations (see, *e.g.*, [HU79]), it is possible to convert $\mathcal{M}$ into a context-free grammar $\mathcal{G}$, whose nonterminal symbols have the form $[pAr]$, for $p, r \in Q$ and $A \in \Gamma$. Then $[qA]$ has a horizontal loop if and only if $L(\mathcal{G}_{|[qAq]})$ contains at least one string besides the empty word. □

If a $[qAp]$-computation $\mathcal{C}$ contains a proper $[qAp]$-subcomputation $\mathcal{C}'$, for the *same* triple $[qAp]$, which starts with stack higher than at the beginning of $\mathcal{C}$, then the pair $(\mathcal{X}, \mathcal{Y})$ where $\mathcal{X}$ is the prefix of $\mathcal{C}$ ending in the first configuration of $\mathcal{C}'$, and $\mathcal{Y}$ is the suffix of $\mathcal{C}$ starting from the last configuration of $\mathcal{C}'$, it is called *vertical loop*. Notice that at the end of $\mathcal{X}$ a nonempty string $A\alpha$ is on the pushdown above the occurrence of A which was on the top at the beginning of $\mathcal{C}$, and such a string is popped off during $\mathcal{Y}$.

In the following sections we shall consider grammars in *binary normal form*, an extension of Chomsky normal form where, in a context-free grammar $\mathcal{G} = \langle V, \Sigma, P, S \rangle$, besides productions $A \to BC$ and $A \to a$, also *unit productions* $A \to B$ and *ε-productions* are allowed.

We remind the reader that unary context-free languages are regular [GR62]. The size cost of the conversion of unary context-free grammar and pushdown automata into equivalent nondeterministic and deterministic finite automata (NFAs and DFAs, resp.) has been investigated in [PSW02]. In the chapter we will use the following small extension of [PSW02, Thms. 4, 6]:

Lemma 8.2. *For each unary context-free grammar $\mathcal{G} = \langle V, \{ a \}, P, S \rangle$ in binary normal form, with v variables, there exist:*

- *an equivalent* NFA *with at most $2^{2^{v-1}} + 1$ states, and*
- *an equivalent* DFA *with less that 2^{v^2} states.*

Proof. By applying a standard construction, from $\mathcal{G}$ we can obtain a grammar $\mathcal{G}'$ in Chomsky normal form, having the same set of variables as $\mathcal{G}$ and generating the same language, with the possible exception of the empty word (if generated by $\mathcal{G}$).

Using Theorems 4 and 6 in [PSW02], we can convert $\mathcal{G}'$ into equivalent finite automata satisfying the bounds on the number of the states given in the statement of the lemma. By inspecting the proofs of those results, it can be observed that there are not transitions entering the initial states of the resulting automata. This allows to safely mark the initial states as accepting, in the case $\mathcal{G}$ generates the empty word, in order to make the resulting automata equivalent to the original grammar $\mathcal{G}$. □

The following result, related to Diophantine equations, will be used in the chapter:

Lemma 8.3 ([MP00, Lemma 2.6])**.** *Let $n, z, i_0, i_1, \ldots, i_s$ be integers with $0 < i_j \leq n$, $j = 0, \ldots, s$, and $z \geq 0$. If the equation $i_0 x_0 + i_1 x_1 + \cdots + i_s x_s = z$ has a solution in natural numbers, then it also has a solution in natural numbers satisfying $i_1 x_1 + \cdots + i_s x_s \leq n^2$.*

8.2 Undecidability and Non-Recursive Bounds

In this section we prove that it cannot be decided whether a PDA accepts in constant height or not. Furthermore, there are no recursive functions that upper limit, with respect to the size of any PDA $\mathcal{M}$ accepting in constant height, the maximal height which is reached by the pushdown store of $\mathcal{M}$ and the number of states of minimal finite automata equivalent to $\mathcal{M}$.

These results are proved by using a technique introduced in [Har67], based on suitable encodings of single-tape Turing machine computations. Roughly, configurations of a such machine $\mathcal{T}$ with state set Q and alphabet Γ are denoted in the standard way as strings from $\Gamma^* Q \Gamma^*$. A computation consisting of m configurations $\alpha_1, \alpha_2, \ldots, \alpha_m$ is encoded as a string of blocks, separated by a delimiter $\$ \notin Q \cup \Gamma$, where the i-th block is α_i when i is odd, α_i^R when i is even (in the following, we use $\alpha_i^{(R)}$ to denote either α_i^R or α_i according to the parity of the index i).

Hence, the (encoding of a) *valid computation* of $\mathcal{T}$ on input w is a string

$$\mathcal{C} = \alpha_1 \$ \alpha_2^R \$ \alpha_3 \$ \alpha_4^R \$ \cdots \$ \alpha_m^{(R)},$$

for some integer $m \geq 1$ such that:

1. $\alpha_i \in \Gamma^* Q \Gamma^*$, *i.e.*, α_i encodes a configuration of $\mathcal{T}$, $i = 1, \ldots, m$;
2. α_1 is the initial configuration on input w;
3. α_{i+1} is reachable in one step from α_i, $i = 1, \ldots, m-1$;
4. α_m is a halting configuration of $\mathcal{T}$.

A *partial valid computation* is defined in a similar way, by dropping Condition 4.

As proved in by Hartmanis, the complement of the set of all valid computations of $\mathcal{T}$ is a context-free language [Har67].

Theorem 8.1. *It is undecidable whether a* PDA *accepts in constant height.*

Proof. We give a reduction from the halting problem. Let $\mathcal{T}$ be a deterministic Turing machine. With an easy modification, we suppose that arbitrarily long computations use arbitrarily large amounts of tape (to this aim, it is sufficient to modify $\mathcal{T}$ by adding to the tape a track where the machine, between any two original consecutive moves, marks a tape cell not visited yet).

By adapting the techniques used by Hartmanis to prove the above mentioned result, we show that the complement of the language $partial(\mathcal{T}, w)$ of partial computations of $\mathcal{T}$ on a given input w is accepted by a PDA $\mathcal{M}_{\mathcal{T},w}$ in the following way. Given $\mathcal{D} = \beta_1\$\beta_2^R\$\cdots\$\beta_r^{(R)}$, with $\beta_i \in (Q \cup \Gamma)^*$, $i = 1, \ldots, r$, to decide whether $\mathcal{D} \in (partial(\mathcal{T}, w))^c$, $\mathcal{M}_{\mathcal{T},w}$ guesses which one among Conditions 1, 2 and 3 is not satisfied. For the first two conditions, the verification of the guess is done by only using the finite control. For the third condition, $\mathcal{M}_{\mathcal{T},w}$ nondeterministically selects one block $\beta_i^{(R)}$, $1 \leq i \leq r$, copies it on the pushdown store and then makes the verification. This is done by scanning the $(i+1)$-th block, if any, and by suitably comparing it with the block saved on the pushdown store. (If $i = r$ then the verification fails.)

We remind the reader that the pushdown height used to accept any input string x is the minimum height in accepting computations on x. Hence, if $\mathcal{D}$ does not satisfy Condition 1 or Condition 2, then it is accepted with pushdown height 0; otherwise, the height is bounded by the length of the first block $\beta_i^{(R)}$ for which Condition 3 is not satisfied, *i.e.*, the block corresponding to the largest i such that $\beta_j = \alpha_j$, for $j = 1, \ldots, i$, where $\alpha_1, \alpha_2, \ldots$ is the (possibly infinite) sequence of configurations in the computation of $\mathcal{T}$ on w.

If $\mathcal{T}$ halts on w in m steps, then the maximum amount of the pushdown store used to accept strings in $(partial(\mathcal{T}, w))^c$ is equal to $|\alpha_m|$. Otherwise, for each arbitrarily large integer h, we can find an index $i > 0$ such that $|\alpha_i| > h$. To accept any string $\alpha_1\$\alpha_2^R\$\cdots\$\alpha_i^{(R)}\$\beta$, with $\beta \in \Gamma^* Q \Gamma^*$ and $\beta \neq \alpha_{i+1}^{(R)}$, $\mathcal{M}_{\mathcal{T},w}$ uses height $|\alpha_i| > h$.

This allows to conclude that $\mathcal{T}$ halts on input w if and only if $\mathcal{M}_{\mathcal{T},w}$ accepts in constant height. Hence, it cannot be decided whether a PDA accepts in constant height. □

We point out that in the restricted case of *unambiguous* PDAs, the property in Theo-

rem 8.1 is decidable [Mal+12]. As a consequence, it is decidable for *deterministic* PDAs.

As already observed at the beginning of the chapter, any PDA $\mathcal{M}$ accepting in constant height h can be converted into an equivalent constant-height PDA. From such a machine, equivalent NFAs and DFAs with a number of states exponential and double exponential in h, respectively, are easily obtained. In the worst case these bounds cannot be reduced [GMP10]. We now show that, however, h cannot be bounded by any recursive function in the size of $\mathcal{M}$.

Theorem 8.2. *For any recursive function $f : \mathbb{N} \to \mathbb{N}$ and for infinitely many integers n there exists a* PDA *of size n accepting in constant height $H(n)$, where $H(n)$ cannot be bounded by $f(n)$.*[1]

Proof. The argument is derived from [MF71, Prop. 7]. Let us consider a busy beaver $\mathcal{BB}_n$ as defined in Example 2.2. As recalled in Theorem 2.1, $\Sigma(n)$ cannot be bounded by any recursive function [Rad62]. Hence, also the maximal length of configurations occurring in a computation of $\mathcal{BB}_n$ cannot be bounded by any recursive function.

Let $\mathcal{C}_n$ be the encoding of the valid computation of $\mathcal{BB}_n$ on ε. By adapting the arguments used to prove Theorem 8.1, we can define a PDA $\mathcal{M}_n$, whose description has a size polynomial in n, which accepts all the strings over $(Q_n \cup \Gamma \cup \{\$\})^*$ different from $\mathcal{C}_n$, and such that each string different from $\mathcal{C}_n$ is accepted using height bounded by the length of the longest configuration occurring in $\mathcal{C}_n$. Since n is fixed, $\mathcal{M}_n$ accepts in constant height. Furthermore, by suitably modifying $\mathcal{C}_n$ (with the same method we applied in the last part of the proof of Theorem 8.1 to a prefix of the string encoding the infinite computation of the machine $\mathcal{T}$ on input w), we can obtain a string that requires height equal to the maximal length of configurations occurring in $\mathcal{C}_n$ to be accepted by $\mathcal{M}_n$.

This allows to conclude that the pushdown height used by $\mathcal{M}_n$ cannot be bounded by any recursive function in the size of $\mathcal{M}_n$. □

The PDA $\mathcal{M}_n$ used to prove Theorem 8.2 accepts the complement of the singleton language $\{ \mathcal{C}_n \}$. This implies that each equivalent deterministic automaton requires more than $|\mathcal{C}_n|$ states. Hence:

[1] We point out that here $H(n)$ is a function of the size of the PDA and *not* of the input.

Corollary 8.1. *There is no recursive function bounding the size blowup from* PDAs *accepting in constant height to finite automata.*

8.3 Constant Height Decidability in the Unary Case

In Section 8.2 we proved that it cannot be decided if a PDA accepts in constant height. This section is devoted to showing that this property turns out to be decidable in the restricted case of PDAs with a one-letter input alphabet. We first give an informal outline of the argument.

Any accepting computation on a sufficiently long input should contain horizontal or vertical loops. The use of vertical loops can lead to computations using unbounded height. However, we prove that if an accepting computation on an input a^ℓ visits a surface pair on which there exists a horizontal loop, then there is another accepting computation for the same input in which almost all occurrences of the vertical loops are replaced by occurrences of such horizontal loop. The number of vertical loops which remain in the resulting computation is bounded by a constant. As a consequence, the amount of pushdown store sufficient to accept a^ℓ is also bounded by a constant. This result is obtained by refining pumping arguments on grammars and the fact that, in the unary case, input symbols commute. In contrast, when all accepting computations on a long string a^ℓ do not visit any surface pair having a horizontal loop, vertical loops and an increasing of the stack cannot be avoided. Hence, the given PDA works in constant height if and only if the cardinality of the language $L_v \setminus L_h$ is finite, where L_h (L_v, resp.) is the set of strings which are accepted by a computation visiting a (not visiting any, resp.) surface pair having a horizontal loop. Since we are considering a unary alphabet, languages L_v and L_h are regular. So the finiteness of their difference is decidable.

To obtain these results, we refine some of the arguments given by Pighizzini, Shallit, and Wang to study the size costs of the transformations of unary context-free grammars and pushdown automata into equivalent finite automata [PSW02].

8.3.1 Loops and grammars

In the following, we consider a grammar $\mathcal{G} = \langle V, \Sigma, P, S \rangle$ in *binary normal form* and we denote by $v = \#V$ the number of its variables.

If T is a parse tree whose root is labeled with a variable $A \in V$ and such that the labels of the leaves, from left to right, form a string $\gamma \in (V \cup \Sigma)^*$, then we write $T : A \stackrel{*}{\Rightarrow} \gamma$. Furthermore, we indicate by $\nu(T)$ the set of variables occurring as labels of the nodes in T. As usual, the *height of a derivation tree* T is the maximum number of edges from the root to a leaf in T.

A *gap tree* T from a variable $A \in V$, also called *A-gap tree*, is a tree corresponding to a nonempty derivation of the form $A \stackrel{+}{\Rightarrow} xAy$, with $x, y \in \Sigma^*$. When $x = y = \varepsilon$, the gap tree T is said to be *trivial*, otherwise, *i.e.*, when it has at least one leaf labeled by a terminal, T is *nontrivial*.

Lemma 8.4. *If $A \stackrel{*}{\Rightarrow} \gamma$, $A \in V$, $\gamma \in (V \cup \Sigma)^+$, then there exists a derivation tree $T : A \stackrel{*}{\Rightarrow} \gamma$ of height at most $(|\gamma| + 1)v$.*

Proof. Given a derivation tree $T : A \stackrel{*}{\Rightarrow} \gamma$ of height $h > (|\gamma| + 1)v$, let $n_1, n_2, \ldots, n_h$ be the sequence of the internal nodes which are encountered on a longest path in T, moving from the leaf to the root. With each node n_k, we associate the pair (A_k, γ_k), where A_k is the variable labeling n_k and γ_k is the string generated by the subtree rooted at n_k.

Hence, for $k = 2, \ldots, h$, γ_{k-1} is a factor of γ_k and $\gamma_h = \gamma$. Considering that γ_1 could be ε, the number of possible different second components in these pairs is bounded by $|\gamma| + 1$.

Since $h > (|\gamma| + 1)v$, this implies that there is a sequence of $v + 1 > \#V$ indices $k, k + 1, \ldots, k + v$, with $1 \leq k \leq h - v$, such that $\gamma_k = \gamma_{k+1} = \ldots = \gamma_{k+v}$. Hence $A_i = A_j$, for some i, j, with $k \leq i < j \leq k + v$, namely, the tree T_1 obtained by removing from the subtree of T rooted at n_j the subtree rooted at n_i is a trivial A_i-gap tree. By removing T_1 from T, *i.e.*, by replacing the subtree rooted at n_j by the subtree rooted at n_i, we obtain a tree $T' : A \stackrel{*}{\Rightarrow} \gamma$ with a smaller number of nodes than T.

We can iterate this process, up to obtain a tree for a derivation $A \stackrel{*}{\Rightarrow} \gamma$ of height bounded by $(|\gamma| + 1)v$. □

Lemma 8.5. *Let $T : A \stackrel{*}{\Rightarrow} \gamma$, $A \in V$, $\gamma \in (V \cup \Sigma)^+$, be a derivation tree.*

1. *Let k be the length of the longest path from the root to a leaf labeled by a terminal symbol, if any. Then γ contains at most 2^{k-1} terminal symbols, and less than 2^{k-1} symbols when γ contains at least one variable.*

2. *If γ contains only terminal symbols,* i.e., *$\gamma \in \Sigma^*$, and the height of T is k, then $|\gamma| \leq 2^{k-1}$.*

3. *If $\gamma = xAy$, $xy \in \Sigma^+$, and T has a minimal number of nodes among all nontrivial A-gap trees, then $|xy| < 2^{2v-1}$.*

Proof. The proof is given by adapting standard properties of parse trees of grammars in Chomsky normal form (see, *e.g.*, [HU79]).

1. The statement can be proved by induction on k. If $k = 1$ then the tree consists only of the root, labeled by A, with one son, labeled by $a \in \Sigma$, where $A \to a$ is a production of $\mathcal{G}$. In this case the statement is trivial. If $k > 1$ then, in any subtree of the root, the longest path to a leaf labeled by a terminal symbol has length at most $k - 1$, so generating, by induction hypothesis, a string containing at most 2^{k-2} terminal symbols. Due to the form of the grammar, the root can have at most 2 subtrees. Hence the number of terminal symbols in γ is bounded by 2^{k-1}. Furthermore, when γ contains one variable, one of the subtrees of the root derives a factor of γ containing such a variable. Hence, by induction, it generates a number of terminals which is strictly less than 2^{k-2}. As a consequence, the number of terminals in γ is less than 2^{k-1}.

2. Consequence of Item 1.

3. Let $n_1, n_2, \ldots, n_k$ be the sequence of the internal nodes on the longest path in T from a leaf labeled by a terminal symbol $a \in \Sigma$ to the root. With each node n_i, $i = 1, \ldots, n$, we associate a pair (A_i, b_i) where $A_i \in V$ and $b_i \in \{ 0, 1 \}$ is 1 if and only if the factor

of γ generated by the subtree rooted at n_i contains the variable A. Hence, the pair associated with n_1 is $(B, 0)$ for some $B \in V$ having the production $B \to a$, while the pair associated with the root n_k is $(A, 1)$.

Suppose $k > 2v$. Then there are two nodes n_i, n_j, with $1 \leq i < j \leq k$ with $(A_i, b_i) = (A_j, b_j)$. By replacing in T the subtree rooted at n_j by the subtree rooted at n_i we obtain an A-gap tree T' which still generates at least one terminal symbol and has less nodes than T, which is a contradiction.

Hence, in T each path connecting the root and a leaf labeled by a terminal symbol should have length at most $2v$, which, according to Item 1, implies $|xy| < 2^{2v-1}$.

□

From now on, let us suppose that $\mathcal{G}$ is unary, *i.e.*, $\Sigma = \{ a \}$. The following modified version of Lemma 2(ii) in [PSW02] is derived from the arguments of the classical "pumping lemma" for context-free languages.

Lemma 8.6. *Let $T : S \stackrel{\star}{\Rightarrow} a^\ell$ be a tree in $\mathcal{G}$. If $\ell > 2^{v^2-1}$ then there exist three integers s, i, j, with $\ell = s + i + j$, $s \geq 0$, and $0 < i + j < 2^{v^2}$, a tree $T_1 : S \stackrel{\star}{\Rightarrow} a^s$, a variable $A \in \nu(T)$, and an A-gap tree $T_2 : A \stackrel{+}{\Rightarrow} a^i A a^j$, such that $\nu(T) = \nu(T_1) \supseteq \nu(T_2)$.*

Proof. We use a combinatorial argument similar to that in the proofs of Lemmas 8.4 and 8.5. Let $n_1, n_2, \ldots, n_e$ be the sequence of the internal nodes which are encountered on a longest path in T, moving from the leaf to the root. With each n_k we associate the pair (A_k, α_k), where $A_k \in V$ is the variable labeling n_k and $\alpha_k \subseteq \nu(T)$ is the set of variables occurring in the subtree rooted at n_k, $k = 1, \ldots, e$. Hence, $\alpha_1 = \{ A_1 \}$, $\alpha_e = \nu(T)$, and $\alpha_{k-1} \subseteq \alpha_k$, $k = 2, \ldots, e$. Notice that we can have at most v different α_k's.

Since $\ell > 2^{v^2-1}$, from Lemma 8.5(2) we get $e > v^2$. Thus, there are more than v consecutive pairs (A_k, α_k) with the same second component and, so, the first components of two of them should coincide. In other words, we can find two nodes n_x and n_y, $0 < x < y \leq v^2 + 1$, such that $(A_x, \alpha_x) = (A_y, \alpha_y)$ and $y - x \leq v$. By replacing in T the subtree rooted at n_y by the subtree rooted at n_x, we get a new tree $T_1 : S \stackrel{\star}{\Rightarrow} a^s$ with $s \leq \ell$. Let T_2

be the gap tree obtained from T by taking as root n_y and by deleting the subtree rooted at n_x. Then $T_2 : A_x \stackrel{+}{\Rightarrow} a^i A_x a^j$, for some integers i, j with $s + i + j = \ell$. Furthermore, $\nu(T) = \nu(T_1) \supseteq \nu(T_2)$.

Since the root of T_2 is n_y, its height is at most $v^2 + 1$. By Lemma 8.5(1) this implies $i + j < 2^{v^2}$.

Finally, we observe that in case $i + j = 0$ and $s = \ell$, we can repeat the same argument after replacing T by T_1. Since the number of nodes in the "new" T is smaller than in the "old" one, by iterating this process, at some point we will finally obtain a tree T_1 producing a shorter string and a gap tree T_2 producing at least one terminal symbol. □

The following lemma will be crucial to obtain our main result. We prove that each long enough string a^ℓ can be derived by pumping a derivation tree of some short string by many occurrences of a *same* gap tree. Furthermore, such a gap tree can be arbitrarily chosen among "small" nontrivial A-gap trees, with A occurring in the derivation of a^ℓ.

Lemma 8.7. *For any derivation tree $T : S \stackrel{\star}{\Rightarrow} a^\ell$ and for any A-gap tree $T_A : A \stackrel{\star}{\Rightarrow} a^i A a^j$, with $0 < i + j < 2^{2v-1}$ and $A \in \nu(T)$, there exists a derivation tree $T' : S \stackrel{\star}{\Rightarrow} a^\ell$ which is obtained by pumping a tree $T_0 : S \stackrel{\star}{\Rightarrow} a^{\ell_0}$ such that $\nu(T_0) = \nu(T)$, $0 \leq \ell_0 \leq 2^{2v^2} - 3 \cdot 2^{v^2-1} + 1$, with $k \geq 0$ occurrences of T_A.*

Proof. If $\ell \leq 2^{2v^2} - 3 \cdot 2^{v^2-1} + 1$, then we take $T_0 = T$, $\ell_0 = \ell$, and $k = 0$. Otherwise, we repeatedly apply Lemma 8.6 to "unpump" the tree T up to find a tree $T_r : S \stackrel{\star}{\Rightarrow} a^{\ell_r}$, with $\ell_r \leq 2^{v^2-1}$ and $\nu(T_r) = \nu(T)$.

Let $\{ i_1, \ldots, i_s \} \subseteq \{ 1, \ldots, 2^{v^2} - 1 \}$ be the set of numbers of terminals that are generated by the gap trees removed during this process. Hence $\ell = \ell_r + i_1 x_1 + \cdots + i_s x_s$, where, for $t = 1, \ldots, s$, $x_t > 0$ is the number of gap trees generating i_t terminal symbols that have been removed to obtain T_r. Let $i_0 = i + j < 2^{2v-1} \leq 2^{v^2}$ be the number of terminals generated by the tree T_A. By Lemma 8.3 (applied with $z = \ell - \ell_r$ and $x_0 = 0$), we can find integers $x'_0, x'_1, \ldots, x'_s \geq 0$ in such a way that $\ell = \ell_r + i_0 x'_0 + i_1 x'_1 + \cdots + i_s x'_s$ and $i_1 x'_1 + \cdots + i_s x'_s \leq (2^{v^2} - 1)^2$. This means that we can pump the tree T_r with a suitable number of occurrences of some of the gap trees removed in the previous process, in

order to get a tree $T_0 : S \stackrel{\star}{\Rightarrow} a^{\ell_0}$, with $\ell_0 = \ell_r + i_1 x'_1 + \cdots + i_s x'_s \leq 2^{v^2-1} + (2^{v^2} - 1)^2 = 2^{2v^2} - 3 \cdot 2^{v^2-1} + 1$ and $\nu(T_0) = \nu(T)$. Furthermore, by pumping T_0 with x'_0 occurrences of T_A, we finally get a tree $T' : S \stackrel{\star}{\Rightarrow} a^{\ell}$. □

8.3.2 Simulating vertical loops by a horizontal loop

From now on, let us consider a fixed PDA $\mathcal{M} = \langle Q, \Sigma, \Gamma, \delta, q_I, Z_0, \{ q_F \} \rangle$. We define the grammar $\mathcal{G} = \langle V, \Sigma, P, S \rangle$, where the elements of V are triples $[qAp]$, with $q, p \in Q$, $A \in \Gamma$, the start symbol S is the triple $[q_I Z_0 q_F]$, and P contains the following productions:

1. $[qAp] \to [qAr][rAp]$, for $q, p, r \in Q$, $A \in \Gamma$;
2. $[qAp] \to [q'Bp']$, for $q, q', p, p' \in Q$, $A, B \in \Gamma$ such that $(q', \texttt{push}(B)) \in \delta(q, \varepsilon, A)$ and $(p, \texttt{pop}) \in \delta(p', \varepsilon, B)$;
3. $[qAp] \to \sigma$, for $q, p \in Q$, $\sigma \in \Sigma \cup \{ \varepsilon \}$, $A \in \Gamma$ such that $(p, -) \in \delta(q, \sigma, A)$;
4. $[qAq] \to \varepsilon$, for $q \in Q$, $A \in \Gamma$.

We point out that the number of variables of $\mathcal{G}$ is $v = (\#Q)^2 \cdot \#\Gamma$. Furthermore, $\mathcal{G}$ is in binary normal form.

The techniques used to show that $\mathcal{G}$ generates the language accepted by $\mathcal{M}$ are very similar to those presented in classical textbooks (see, *e.g.*, [HU79]) to prove the correctness of the standard transformation of PDAs into CFGs. In particular, it can be shown that each triple $[qAp]$ generates the language $L_{[qAp]}$ of all strings which are consumed in $[qAp]$-computations.

Since we are interested in the amount of stack used by $\mathcal{M}$, we state such equivalence in a stronger form, which also considers the use of the stack in computations.

In particular, we relate the stack increment to the *unit production height* which, for a derivation tree T of the above grammar $\mathcal{G}$, is defined as the maximum number of edges corresponding to unit productions in a path from the root to a leaf of T.

Lemma 8.8. *For any $x \in \Sigma^*$, $q, p \in Q$, $A \in \Gamma$, $h \in \mathbb{N}$, there exists a derivation tree $T : [qAp] \stackrel{\star}{\Rightarrow} x$ with unit production height h if and only if there exists a $[qAp]$-computation $\mathcal{C}$ on x with stack increment h.*

Proof. Let $T : [qAp] \Rightarrow x$, for some $k > 0$, be a derivation tree with unit production height h. We prove by induction on k that there exists a $[qAp]$-computation $\mathcal{C}$ on x with stack increment h.

If $k = 1$ then the tree contains only the root and one leaf and it corresponds to the use of one of the productions of the form 3 or 4. So the statement is trivial.

If $k > 1$ then the production used at the root level of T is either of the form 1 or of the form 2.

In the first case, we have $[qAp] \rightarrow [qAr][rAp]$, for some $r \in Q$, the root of T has left subtree $T' : [qAr] \Rightarrow x'$ and a right subtree $T'' : [rAp] \Rightarrow x''$, for some $k', k'' > 0$ with $k' + k'' = k - 1$, $x', x'' \in \Sigma^*$, $x'x'' = x$. Then $h = \max\{ h', h'' \}$, where h' and h'' are the unit production heights of T' and T'', respectively. According to the induction hypothesis, there exist a $[qAr]$-computation on x' and a $[rAp]$-computation on x'' with stack increment h' and h'', respectively. By concatenating these two computations, we obtain a $[qAp]$-computation on x in which the stack increment is $\max\{ h', h'' \}$, namely h.

In case the production applied to the root of T is $[qAp] \rightarrow [q'Bp']$, let $T' : [q'Bp'] \Rightarrow x$ be the subtree of T rooted at the only son of the root. Since in T at the top level a unit production is used, the unit production height on a path from the root of T' is $h - 1$. Even in this case, from the induction hypothesis we obtain a $[q'Bp']$-computation on x with stack increment $h - 1$. By adding to this computation the initial push and the final pop from which the production $[qAp] \rightarrow [q'Bp']$ is defined, we obtain a $[qAp]$-computation on x with stack increment h.

Conversely, let us consider a $[qAp]$-computation $\mathcal{C}$ on x with stack increment h. We proceed on the number k of steps of $\mathcal{C}$.

If $k = 1$ then the computation $\mathcal{C}$ does not make any stack increment and it can only correspond to a one-step derivation consisting of a production of the form 3 or 4. So the

statement is trivial.

If $k > 1$ then we consider two cases, depending on whether or not at some configuration in $\mathcal{C}$, after the first and before the last configuration, the stack is at the same height than at the beginning and at the end of $\mathcal{C}$.

- If such configuration exists, then we split $\mathcal{C}$ at that configuration into a $[qAr]$-computation $\mathcal{C}'$ and a $[rAp]$-computation $\mathcal{C}''$, for some $r \in Q$, consuming some x', x'', with $x'x'' = x$ and with stack increment h', h'', respectively. Then the stack increment in $\mathcal{C}$ is $\max\{h', h''\}$. Using the induction hypothesis, we find two trees T' and T'' corresponding to such computations, with unit production heights h' and h'', respectively. We can suitably combine T' and T'', using a production of form 1, in order to obtain a tree T which derives x and has $\max\{ h', h'' \}$ height.

- If such configuration does not exist, then the computation of $\mathcal{C}$ should start with a push of a symbol B which is removed in the last step. Let $\mathcal{C}'$ be a $[q'Br']$-subcomputation of $\mathcal{C}$ which is obtained by removing the first and the last step. If h is the stack increment in $\mathcal{C}$, then the stack increment in $\mathcal{C}'$ is $h - 1$. Let $T' : [q'Br'] \stackrel{\star}{\Rightarrow} x$ be the tree corresponding to $\mathcal{C}'$, obtained according to the induction hypothesis. Its unit production height is $h - 1$. The tree T which is obtained by taking T' as only subtree of a root with label $[qAr]$, derives x and has unit production height h.

□

As a consequence of Lemma 8.8 we get:

Corollary 8.2. *For any integer $h \geq 0$, a string x is accepted by $\mathcal{M}$ using pushdown height h if and only if there is a derivation tree T of x in $\mathcal{G}$ with unit production height h.*

Combining Corollary 8.2 with Lemma 8.4, we get the following upper bound for the height of the pushdown store necessary to accept a string x:

Lemma 8.9. *If $x \in \Sigma^*$ is accepted by $\mathcal{M}$, then $h(x) \leq (|x| + 1)v$, where $v = (\#Q)^2 \cdot \#\Gamma$.*

Proof. By contradiction, suppose that each computation of $\mathcal{M}$ accepting x uses pushdown height greater than $(|x|+1)v$. As a consequence of Corollary 8.2, each derivation tree of x in $\mathcal{G}$ has unit production height, and so height, greater than $(|x|+1)v$, which is a contradiction to Lemma 8.4. □

Let us go back to the case of unary pushdown automata. Hence, from now on let $\mathcal{M} = \langle Q, \{ a \}, \Gamma, \delta, q_I, Z_0, \{ q_F \} \rangle$ be a fixed *unary* PDA. Using Lemma 8.8, we can reformulate Lemma 8.7 in terms of pushdown automata. Roughly, we can say that for each computation $\mathcal{C}$ accepting a "long" input, there is another computation accepting the same input, which is obtained by pumping a suitable computation $\mathcal{C}_0$, chosen from a finite set, with a repeated pattern which is arbitrarily selected from another finite set that depends on $\mathcal{C}_0$. We will use this property to replace, in any accepting computation $\mathcal{C}$, almost all the vertical loops with many occurrences of a horizontal loop, in the case a surface pair $[rB]$ having a horizontal loop occurs in $\mathcal{C}$. In this way, we shall be able to obtain an accepting computation on the same input using a bounded amount of pushdown storage.

Theorem 8.3. *Let $\mathcal{C}$ be an accepting computation on input a^ℓ which visits a surface pair $[rB]$ having a horizontal loop. Then there exists another accepting computation on a^ℓ which uses pushdown height smaller than $2^{2v^2+\log_2 v}$.*

Proof. Let $\mathcal{G}$ be the above defined grammar, obtained from $\mathcal{M}$. First, we observe that if $\mathcal{C}$ visits the surface pair $[rB]$ then there exists a derivation tree $T : S \stackrel{*}{\Rightarrow} a^\ell$ with $[rBr] \in \nu(T)$. In fact, one of the triples $[rBs]$ or $[sBr]$ for some $s \in Q$ should appear in the derivation tree corresponding to $\mathcal{C}$. Since $\mathcal{G}$ contains the productions $[rBs] \to [rBr][rBs]$, $[sBr] \to [sBr][rBr]$ and $[rBr] \to \varepsilon$, we can suitably modify the tree in order to introduce one occurrence of $[rBr]$, without changing the derived string.

Now we select a "small" $[rBr]$-gap tree $T_{[rBr]}$ deriving a nonempty string, *i.e.*, $T_{[rBr]} : [rBr] \stackrel{*}{\Rightarrow} a^i[rBr]a^j$, with $0 < i + j < 2^{2v-1}$. We prove that such a gap tree should exist. In fact, since $[rB]$ has a horizontal loop, the language $L_{[rBr]}$ should contain at least a nonempty string $w \in a^*$. Hence, $[rBr] \stackrel{*}{\Rightarrow} w$. Furthermore, the tree corresponding to the

derivation

$$[rBr] \stackrel{\star}{\Rightarrow} [rBr][rBr] \stackrel{\star}{\Rightarrow} w[rBr]$$

is a $[rBr]$-gap tree. Hence, from Lemma 8.5(3), it follows that there exists also a $[rBr]$-gap tree $T_{[rBr]} : [rBr] \stackrel{\star}{\Rightarrow} a^i[rBr]a^j$, with $0 < i + j < 2^{2v-1}$.

According to Lemma 8.7, we can obtain another tree $T' : S \stackrel{\star}{\Rightarrow} a^\ell$ by pumping a tree $T_0 : S \stackrel{\star}{\Rightarrow} a^{\ell_0}$, such that $\nu(T_0) = \nu(T)$, $0 \leq \ell_0 \leq 2^{2v^2} - 3 \cdot 2^{v^2-1} + 1$, with $k \geq 0$ occurrences of $T_{[rBr]}$.

We observe that in the tree T', some of the k occurrences of $T_{[rBr]}$, say t, could be nested, possibly giving a stack height in the corresponding computation which linearly increases with k. To fix this problem, we modify T' as we now describe (see Figure 8.1).

Let u be a node of T_0 labeled by $[rBr]$ and T_u be the subtree of T_0 rooted at u, such that T_0 is pumped starting from u with t nested occurrences of $T_{[rBr]}$, $1 < t \leq k$. We rearrange these t occurrences of $T_{[rBr]}$ in a sequence by inserting, starting from node u, a subtree corresponding to a derivation $[rBr] \stackrel{\star}{\Rightarrow}^t$ obtained by using $t - 1$ times the production $[rBr] \rightarrow [rBr][rBr]$. To each leaf of this subtree we append one occurrence of the $[rBr]$-gap tree $T_{[rBr]}$. Finally, to the leaf labeled $[rBr]$ of the first occurrence of $T_{[rBr]}$ we append the tree T_u, and to each of the remaining $t - 1$ leaves labeled $[rBr]$ we append one leaf labeled with the empty word (we remind the reader that $[rBr] \rightarrow \varepsilon$ is a production of $\mathcal{G}$).

Let T'' be the tree obtained after this modification, which still generates a^ℓ. Using Corollary 8.2 we now estimate the amount of pushdown store used in the computation $\mathcal{C}''$ corresponding to T'', by calculating the unit production height of T'', which is bounded by the maximum number h_0 of edges corresponding to unit productions in any path in T_0 plus the maximum number h_1 of such edges in any path in $T_{[rBr]}$ which, in turn, are bounded by the height of T_0 and $T_{[rBr]}$, respectively. Using Lemma 8.4, we get $h_0 \leq (\ell_0 + 1)v$ and $h_1 \leq (i + j + 2)v$ (we remind the reader that the tree $T_{[rBr]}$ generates a string of length $i + j + 1$). Hence, the height of the pushdown is bounded by $h_0 + h_1 \leq (\ell_0 + i + j + 3)v$. Considering the bounds on ℓ_0 and $i + j$, we obtain $\ell_0 + i + j + 3 <$

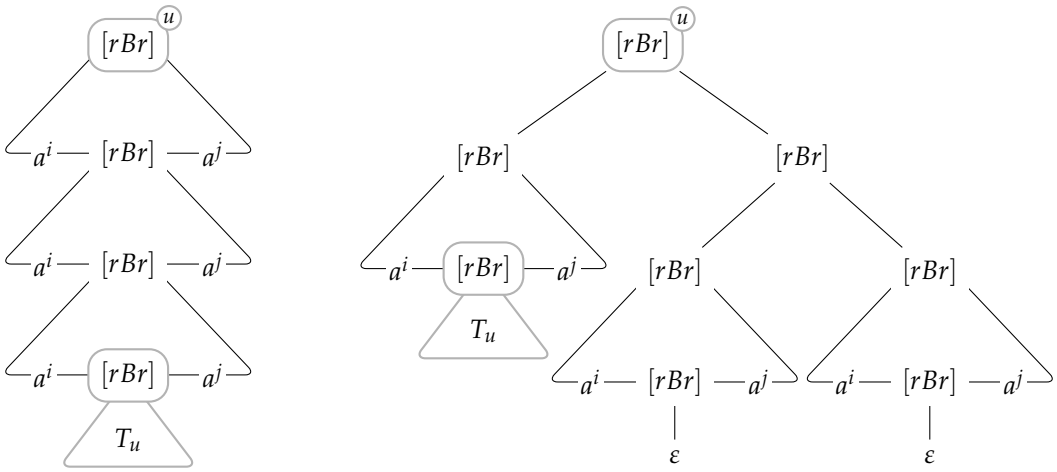

Figure 8.1: On the left, the portion of the tree T_0 from the node u pumped by using t repetitions of the tree $T_{[rBr]}$. These occurrences are rearranged by using $t-1$ times the production $[rBr] \to [rBr][rBr]$, as shown on the right, in order to avoid a linear increase of the stack in the number of the repetitions of $T_{[rBr]}$. (In the figure $t = 3$.)

$2^{2v^2} - 3 \cdot 2^{v^2-1} + 2^{2v-1} + 4$.

For $v \geq 2$, it can be verified that $-3 \cdot 2^{v^2-1} + 2^{2v-1} < -4$. Hence, $h_0 + h_1 < 2^{2v^2} \cdot v = 2^{2v^2+\log_2 v}$.

In the case $v = 1$, the PDA $\mathcal{M}$ can have only one state q, which is both initial and final, and only one pushdown symbol Z_0. Since in the form we are considering for PDAs transitions consuming input symbols do not change the stack (*cf.* Section 2.2.4), the only possibility to read an input symbol is that of having the transition $(q, -) \in \delta(q, a, Z_0)$. If this is the case, then any string in a^* can be accepted by a computation which does not use the pushdown store, *i.e.*, using height 0. Otherwise, $\mathcal{M}$ accepts the empty language and so, by definition, it accepts in height 0. □

8.3.3 Vertical increase without horizontal loops

We now evaluate the increase of the stack in computations that do not use horizontal loops, *i.e.*, between any two repetitions of a same surface pair $[rB]$ at the same height either no input is consumed or there is at least one configuration with lower stack height.

Lemma 8.10. *Let $\mathcal{C}$ be a $[qAp]$-computation on a^ℓ with stack increment bounded by h and without horizontal loops. Then $\ell \leq (\#Q-1)^{h+1}$.*

Proof. We give the proof by induction on h. Let h_0 be the stack height at the beginning and at the end of $\mathcal{C}$. We preliminary observe that since in $\mathcal{C}$ the stack height cannot be lower than h_0 and there are no horizontal loops, between any two repetitions of a same state at stack height h_0 no input symbols can be consumed. Hence, we can remove from $\mathcal{C}$ the part between the two repetitions of such a state, to obtain a shorter $[qAp]$-computation on the same input having stack increment bounded by h. By iterating this process, we finally get $\mathcal{C}$ with at most n configurations at stack height h_0.

If $h = 0$, *i.e.*, the stack height is never incremented, then $\mathcal{C}$ consists of less than $\#Q$ moves. Hence $\ell \leq \#Q-1$. Otherwise, we decompose $\mathcal{C}$ in $k < \#Q$ subcomputations $\mathcal{C}_1, \ldots, \mathcal{C}_k$, where, for $i = 1, \ldots, k$, $\mathcal{C}_i$ starts with a push of a symbol, which is popped off the stack only in the last move of $\mathcal{C}_i$. Let $\mathcal{C}_i'$ be the subcomputation obtained by removing from $\mathcal{C}_i$ the first and the last move and let a^{ℓ_i} be the input consumed during it. Then the stack increment in $\mathcal{C}_i'$ is at most $h-1$. By induction hypothesis, this implies $\ell_i \leq (\#Q-1)^h$. Since push and pop moves do not consume input symbols, we get that $\ell \leq k(\#Q-1)^h \leq (\#Q-1)^{h+1}$. □

As a consequence of Lemma 8.10, the recognition of arbitrarily long strings without making use of horizontal loops requires unbounded stack height.

8.3.4 Decidability

Using the tools we developed so far, we are now able to prove the main result of this section:

Theorem 8.4. *Let $\mathcal{M}$ be a unary* PDA *with n states and m pushdown symbols. Then $\mathcal{M}$ accepts in constant height if and only if it accepts in height smaller than $2^{18v^2+\log_2 v+\log_2 3} = 3v \cdot 2^{18v^2}$, where $v = n^2m$.*

Proof. Let L be the language accepted by $\mathcal{M}$. We also consider the following two languages L_h and L_v:

- L_h is the set of strings accepted by the computations of $\mathcal{M}$ which visit at least one surface pair having a horizontal loop.
- L_v is the set of strings accepted by the computations of $\mathcal{M}$ which visit only surface pairs that do not have horizontal loops.

Clearly, the language L accepted by $\mathcal{M}$ is the union of L_h and L_v.

According to Theorem 8.3, all strings in L_h are accepted in constant height. More precisely, from $\mathcal{M}$ we can build a unary PDA $\mathcal{M}_h$ which accepts L_h by simulating $\mathcal{M}$ and by accepting when the simulated computation is accepting and visits at least one surface pair having a horizontal loop, which can be decided according to Lemma 8.1.

To implement $\mathcal{M}_h$, we double the cardinality of the state set, in order to remember if some surface pair having an horizontal loop has been reached during the computation, *i.e.*, for each state q we create a copy q', where q' is used each time $\mathcal{M}$ reaches q in any configuration which occurs *after* visiting a surface pair having a horizontal loop. Hence, the final state of $\mathcal{M}_h$ is q'_F. From $\mathcal{M}_h$ we can obtain an equivalent grammar in binary normal form with $(2n)^2m$ variables. However, in such a grammar, the triples $[q'Ap]$, where q and p are states of $\mathcal{M}$, cannot generate any string (in fact, once a pair having a horizontal loop is reached, the computation of $\mathcal{M}_h$ can only visit states in the copy of Q). This allows to reduce the number of variables to $3n^2m = 3v$. According to Theorem 8.3, each string in L_h can be accepted using height smaller than $2^{2(3v)^2+\log_2(3v)} = 2^{18v^2+\log_2 v+\log_2 3}$.

If the set $L_v \setminus L_h$ is infinite, then it should contain arbitrarily long strings; by Lemma 8.10, an arbitrarily high stack is required to accept them.

Otherwise, when $L_v \setminus L_h$ is finite, $\mathcal{M}$ accepts in constant height, which is bounded by the maximum between the height used to accept strings in L_h and the height used to accept strings $L_v \setminus L_h$. To estimate the last amount, first we notice that L_v is accepted by a PDA $\mathcal{M}_v$ which can be obtained by just removing from $\mathcal{M}$ all the transitions defined

from surface pairs $[rB]$ having horizontal loops. Hence, L_v is generated by a context-free grammar in binary normal form with $v = n^2m$ variables. According to Lemma 8.2, from $\mathcal{M}_h$ and $\mathcal{M}_v$ we obtain equivalent DFAs with less than 2^{9v^2} and 2^{v^2} states, respectively. From them, using a standard product construction, we can obtain a DFA with less than 2^{10v^2} states accepting $L_v \setminus L_h$. Since such a language is finite, the length of each string in it is less than the number of states of such a DFA, *i.e.*, it is bounded by 2^{10v^2}. By Lemma 8.9, this implies that each string in $L_v \setminus L_h$ is accepted using height bounded by $v2^{10v^2}$, which is lower than the bound we obtained for strings in L_h. By summarizing, we can conclude that if $\mathcal{M}$ accepts in constant height, then it accepts in height smaller than $2^{18v^2+\log_2 v+\log_2 3}$. □

Corollary 8.3. *It is decidable whether a unary* PDA *accepts in constant height.*

8.4 Size versus Height in the Unary Case

The arguments used in Section 8.3 to prove that it is decidable whether a unary PDA accepts in constant height give an exponential upper bound for the maximum stack height, with respect to the size of a PDA working in constant height (see Theorem 8.4). In this section we prove that such an exponential bound cannot be reduced.

To prove this result, we shall make use of some modifications of the PDA described in the following example.

Example 8.1. Let us consider the family of languages $(U_k)_{k=1}^{\infty}$ such that, for each $k > 0$, $U_k = \{ a^{2^k} \}$. A deterministic PDA $\mathcal{A}_k$ for U_k might work as follows. The automaton can exploit its pushdown to implement the recursive function

$$f(i) = \begin{cases} 1 & \text{if } i = 0, \\ 2f(i-1) & \text{otherwise,} \end{cases}$$

in order to read $f(k) = 2^k$ input symbols. To this aim it uses the state set $Q = \{ q, r, p \}$, and the pushdown alphabet $\Gamma = \{ A_0, A_1, \ldots, A_k, B_1, \ldots, B_k \}$.

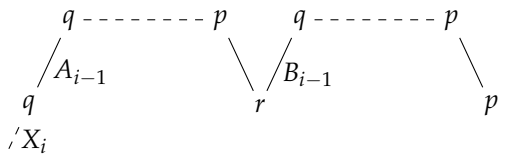

Figure 8.2: The evolution of the pushdown store of $\mathcal{M}_k$ in a $[qX_ip]$-computation, where the symbol X_i which is initially on the top of the stack is either A_i, $i = 1, \ldots, k$, or B_i, $i = 1, \ldots, k-1$. The horizontal dashed lines should be replaced by a move reading one input symbol, when $i - 1 = 0$, and by the same pattern, in the other cases.

One call to $f(i)$ is implemented by a $[qX_ip]$-computation, with $X \in \{ A, B \}$. For $i = 0$, such a computation consists of one move which reads one input symbol (Transitions 1 or 2 below). Otherwise, the computation is split into two parts, both consuming 2^{i-1} input symbols, as depicted in Figure 8.2:

- a $[qX_ir]$-computation that activates, by a recursive call, one $[qA_{i-1}p]$-computation (Transitions 3 or 4, and 7),
- a $[rX_ip]$-computation that activates, by a recursive call, one $[qB_{i-1}p]$-computation (Transitions 5 or 6, and 8).

In this way, a $[qA_kp]$-computation consumes the string a^{2^k}. Hence, to recognize U_k the automaton starts in the state q with A_k on the stack and accepts in the state p.

Formally, $\mathcal{A}_k = \langle Q, \{ a \}, \Gamma, \delta, q, A_k, \{ p \} \rangle$, where the transitions are:

1. $\delta(q, a, A_0) = (p, -)$;
2. $\delta(q, a, B_0) = (p, -)$;
3. $\delta(q, \varepsilon, A_i) = (q, \texttt{push}(A_{i-1}))$, for $i = 1, \ldots, k$;
4. $\delta(q, \varepsilon, B_i) = (q, \texttt{push}(A_{i-1}))$, for $i = 1, \ldots, k-1$;
5. $\delta(r, \varepsilon, A_i) = (q, \texttt{push}(B_{i-1}))$, for $i = 1, \ldots, k$;
6. $\delta(r, \varepsilon, B_i) = (q, \texttt{push}(B_{i-1}))$, for $i = 1, \ldots, k-1$;

7. $\delta(p, \varepsilon, A_i) = (r, \mathtt{pop})$, for $i = 0, \ldots, k-1$;

8. $\delta(p, \varepsilon, B_i) = (p, \mathtt{pop})$, for $i = 0, \ldots, k-1$.

We point out that, analogously to what happens for NSE grammars, also in this case the size of $\mathcal{A}_k$ is linear in the parameter k, while the minimum DFA accepting L_k has $2^k + 1$ states. ■

We now present the main result of this section:

Theorem 8.5. *For each integer $k > 0$ there exists a PDA $\mathcal{M}_k$ having a size linear in k and accepting in height which is constant with respect to the input length but exponential in k.*

Proof. For each integer $k > 0$, let us consider two automata $\mathcal{A}'_k$ and $\mathcal{A}''_k$, accepting the languages $\{ a^{2^k} \}^*$ and $\{ a^{2^k+1} \}^*$, respectively, obtained by modifying the automaton $\mathcal{A}_k$ of Example 8.1 as follows:

- $\mathcal{A}'_k$ is obtained by adding to $\mathcal{A}_k$ the transition $\delta(p, \varepsilon, A_k) = (q, -)$ and by choosing q as final state. This allows $\mathcal{A}'_k$ to recognize $\{ a^{2^k} \}^*$ with pushdown height k, using 3 states and a pushdown alphabet of size $2k + 1$. We point out that, from such a definition, each accepting computation of $\mathcal{A}'_k$ visits the surface pair $[qA_k]$ which has a horizontal loop.

- $\mathcal{A}''_k$ guesses — at the beginning of the computation — how many repetitions of the word a^{2^k+1} are concatenated in the input word. This is done, in a preliminary phase, by pushing one occurrence of the symbol A_k on the store for each guessed repetition (Transitions 9 below). Then, for any such occurrence, $\mathcal{A}''_k$ makes the following operations:

  - it reads one a from the input (Transition 10),
  - it simulates one execution of $\mathcal{A}_k$, using Transitions 1–8 in Example 8.1,
  - it pops the symbol A_k off the pushdown (Transition 11).

Formally, $\mathcal{A}''_k = \langle Q'', \{ a \}, \Gamma \cup \{ Z_0 \}, \delta'', q_I, Z_0, \{ s \} \rangle$, where $Q'' = Q \cup \{ q_I, s \}$, and δ'' is a copy of δ with the addition of the following nondeterministic transitions:

9. $\delta''(q_I, \varepsilon, X) = \{ (q_I, \texttt{push}(A_k)), (s, -) \}$, for $X \in \{ Z_0, A_k \}$;
10. $\delta''(s, a, A_k) = \{ (q, -) \}$,
11. $\delta''(p, \varepsilon, A_k) = \{ (s, \texttt{pop}) \}$.

Notice that $\mathcal{A}''_k$ has 5 states and $2k + 2$ pushdown symbols. Furthermore, the pushdown height used to accept the string $a^{\beta(2^k+1)}$ is $\beta + k$. So, $\mathcal{A}''_k$ does not accept in constant height.

It is easy to see that the automaton $\mathcal{M}_k$ obtained by concatenating the automata $\mathcal{A}'_k$ and $\mathcal{A}''_k$ using standard techniques (after renaming the states in such a way that the two sets of states are disjoint) recognizes the language

$$H_k = \left\{ a^t \mid t = \alpha 2^k + \beta(2^k + 1),\ \alpha, \beta \geq 0 \right\}$$

and has 8 states and a pushdown alphabet of $2k + 2$ symbols.

By construction, the first part of each accepting computation of $\mathcal{M}_k$ is an accepting computation of $\mathcal{A}'_k$ which, as above observed, visits a surface pair having a horizontal loop. Hence, from Theorem 8.3 it follows that $\mathcal{M}_k$ accepts in constant height, with respect to the input length.

We now prove that a height exponential in k is necessary.

Let us consider the string $a^t \in H_k$ obtained by choosing $\alpha = 0$ and $\beta = 2^k - 1$, namely, $t = (2^k - 1)(2^k + 1) = 2^{2k} - 1$. We are going to prove that there is only one accepting computation on a^t.

To this aim, we observe that, due to the structure of $\mathcal{M}_k$, for each accepting computation on a^t, there should exist two integers $\alpha', \beta' \geq 0$, such that $t = 2^{2k} - 1 = \alpha' 2^k + \beta'(2^k + 1)$, from which $2^{2k} - 1 - \beta' = 2^k(\alpha' + \beta')$ and $2^{2k} - 2^k(\alpha' + \beta') = \beta' + 1$. So, 2^k should divide $\beta' + 1$. The only possible solution of $t = \alpha' 2^k + \beta'(2^k + 1)$ is obtained by taking $\alpha' = \alpha = 0$ and $\beta' = \beta = 2^k - 1$. In fact, this solution corresponds to the smallest $\beta' \geq 0$ such that 2^k divides $\beta' + 1$, while $\beta'(2^k + 1) > t$ for any larger multiple β' of 2^k.

This allows to conclude that the only accepting computation on a^t is the one in which the simulation of $\mathcal{A}''_k$ uses height $\beta + k$, with $\beta = 2^k - 1$.

Hence, to accept a^t an exponential height, with respect the size of $\mathcal{M}_k$, is necessary. □

8.5 An Optimal Lower Bound for Non-Constant Height

In this section we turn our attention to PDAs accepting in non-constant height. First of all, we mention that each nondeterministic Turing machine, with a two-way read-only input tape, which accepts in $o(\log \log m)$ space, where m is the input length and the space is measured by considering the portion of an auxiliary work tape used during the less expensive computation, actually uses only a constant amount of space [Alb85]. As a corollary, the height of the pushdown store in any PDAs accepting in non-constant height should grow at least as the function $\log \log m$. Furthermore, this lower bound is optimal [Bed+16].

We show that in the unary case the optimal bound increases to a logarithmic function.

Let us start by proving the lower bound:

Theorem 8.6. *Let $\mathcal{M}$ be a unary* PDA *using height $h(m)$. Then either $h(m)$ is bounded by a constant or there exists $c > 0$ such that $h(m) \geq c \log m$ infinitely often.*

Proof. According to the proof of Theorem 8.4, if $h(m)$ is not constant, then there exist infinitely many strings in $L_v \setminus L_h$ that are accepted only by computations that use vertical loops and do not visit surface pairs having horizontal loops. Let $\mathcal{M}_v$ be the PDA accepting L_v (proof of Theorem 8.4), which has the same set of states Q and the same pushdown alphabet Γ of $\mathcal{M}$, but that does not visit surface pairs having horizontal loops in accepting computations.

Given $a^m \in L_v \setminus L_h$, let us consider the PDA $\mathcal{M}_{h(m)}$ obtained by bounding the height of the pushdown of $\mathcal{M}_v$ to $h(m)$. To this aim, the pushdown alphabet is extended in order to keep track of the pushdown height, together with each symbol pushed on the stack. Hence, since the only symbol that appears at height 0 is Z_0, the cardinality of the

pushdown alphabet of $\mathcal{M}_{h(m)}$ is bounded by $\#\Gamma \cdot h(m) + 1$. According to the construction in Section 8.3.2, $\mathcal{M}_{h(m)}$ can be converted into an equivalent grammar in binary normal form with $(\#Q)^2 \cdot (\#\Gamma \cdot h(m) + 1)$ variables, from which, using Lemma 8.2, we can obtain an NFA $\mathcal{N}_{h(m)}$ which is equivalent to $\mathcal{M}_{h(m)}$, and whose number of states is $2^{O(h(m))}$.

Since $\mathcal{M}_{h(m)}$ has stack height bounded by $h(m)$, it cannot have vertical loops. Furthermore, since accepting computations of $\mathcal{M}_v$ do not use surface pairs with horizontal loops, also accepting computations of $\mathcal{M}_{h(m)}$ do not use horizontal loops. This allows to conclude that the language accepted by $\mathcal{M}_{h(m)}$ is finite. Thus, in $\mathcal{N}_{h(m)}$, which accepts the same language, the string a^m is accepted by a path without any repeated state. Hence $\mathcal{N}_{h(m)}$ must have more than m states.

This allows to conclude that $2^{O(h(m))} > m$, thus implying the existence of a constant c such that $h(m) \geq c \log m$ infinitely often. □

We now prove a matching upper bound by presenting an overcomplicated device accepting all unary strings:

Theorem 8.7. *There exists a unary* PDA *accepting every word a^m, $m > 0$, using pushdown height exactly $\lfloor \log_2 m \rfloor + 1$ and the empty word using height 0.*

Proof. Consider the PDA $\mathcal{A} = \langle Q, \{ a \}, \Gamma, \delta, q_I, Z_0, q_F \rangle$, where $Q = \{ q_I, q_1, q_2, q_F \}$, $\Gamma = \{ Z_0, 0, 1 \}$, and the transition function δ is defined as follows:

1. $\delta(q_I, \varepsilon, X) = (q_F, -)$, for $X \in \Gamma$;
2. $\delta(q_I, \varepsilon, X) = (q_I, \texttt{push(0)})$, for $X \in \Gamma$;
3. $\delta(q_F, \varepsilon, 0) = (q_1, \texttt{pop})$;
4. $\delta(q_1, a, X) = (q_2, -)$, for $X \in \Gamma$;
5. $\delta(q_2, \varepsilon, X) = (q_I, \texttt{push(1)})$, for $X \in \Gamma$;
6. $\delta(q_F, \varepsilon, 1) = (q_F, \texttt{pop})$.

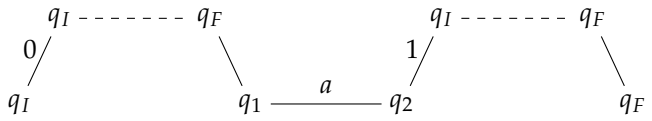

Figure 8.3: The evolution of the pushdown store of $\mathcal{A}$ during the recursive subroutine leading from q_I to q_F, when recursive calls are made. The dashed lines should be replaced either by an ε-move or, recursively, by the same pattern.

As a first observation, we point out that from the initial state q_I it is possible to reach the final state q_F with the same pushdown height by using a subroutine implementing a recursive strategy: either an ε-move is performed (Transitions 1), or two recursive calls of the same subroutine, with one read operation between them, are executed. This is done, as depicted in Figure 8.3, by pushing 0 on the stack, while activating the first recursive call (Transition 2). When such a recursive call ends, 0 is popped off the stack (Transition 3), a symbol a is read from the input (Transition 4), and then 1 is pushed while activating the second call (Transitions 5). Finally, when such a call ends, the symbol 1 is popped off the pushdown (Transitions 6).

It is possible to notice that each string in a^* is accepted by $\mathcal{A}$.

In order to state how the length of a word and the height of the pushdown used for accepting it are related, let us calculate the maximal length $\ell(h)$ of strings consumed by $[q_I X q_F]$-computations with stack increment h, for $X \in \Gamma$. We point out that the moves made during such computations do not depend on the symbol X, hence also $\ell(h)$ does not depend on X.

According to the recursive subroutine implemented by $\mathcal{A}$ (see also Figure 8.3), we can write the following recurrence:

$$\ell(h) = \begin{cases} 0 & \text{if } h = 0, \\ 2\ell(h-1) + 1 & \text{otherwise,} \end{cases}$$

which has solution $\ell(h) = 2^h - 1$. As a consequence, pushdown height h is necessary and sufficient to accept all strings of length m, with $2^{h-1} \leq m < 2^h$. Hence, for $m > 0$, the string a^m is accepted in height $\lfloor \log_2 m \rfloor + 1$. □

9 Future Work

In this dissertation we presented some results related to the area of descriptional complexity, and, in this regard, we focused on the class of regular languages. To conclude the work, we discuss possible directions for future research in this field.

First of all, it is worth to mention that there exist other models characterizing the class of regular languages besides the ones analyzed in this thesis. One example are the well-known *regular expressions*, widely discussed in classical textbooks (see, *e.g.* [HU79, Sha08]).

From regular expressions we can derive a more succinct representation of regular languages, by using *straight line programs*, namely programs representing directed acyclic graphs, whose internal nodes represent the basic regular operations (*i.e.*, union, concatenation, and star). Descriptional complexity of straight line programs has been analyzed and it has been proved that they are polynomially related in size with constant height pushdown automata [GMP10].

As widely discussed, the question posed by Sakoda and Sipser in 1978 about the elimination of nondeterminism from two-way automata is still open. We plan to continue the investigation on this question by considering models that have the same computational power of finite automata and by studying the relations in sizes between these nondeterministic devices and finite automata by deterministic ones.

For example, as remarked by Pighizzini, at the moment direct simulations of 1-limited

automata by *deterministic* 1-limited automata and by two-way deterministic automata are not known [Pig19]. It could be interesting to study if simulating unary and non-unary 1-limited automata by two-way (instead of one-way) deterministic finite automata the cost reduces from a double exponential to a simple exponential.

It could be also interesting to study "relaxed" versions of the problem of Sakoda and Sipser, in which the simulating machine is a deterministic 1-limited automaton (*i.e.*, a deterministic two-way automaton with the capability of rewriting the contents of tape cells during the first visit).

Moreover, following the research line started in [GGP14], it could be deepen the investigation on the Sakoda and Sipser problem in case of simulated devices that perform a limited use of nondeterminism. In this regard, it is possible to consider several restrictions like, for example, on the number of nondetermistic choices along the computation (see, *e.g.* [HK19]), or on the number of total, or accepting, computations (also known as *degree of ambiguity* [Leu05, HSS17, KS19]).

Bibliography

[ACC03] Stefan Andrei, Salvador Valerio Cavadini, and Wei-Ngan Chin:
A New Algorithm for Regularizing One-Letter Context-Free Grammars. *Theoretical Computer Science* 306(1-3), pp. 113–122, 2003. DOI: 10.1016/S0304-3975(03)00215-9.

[AGV02] Marcella Anselmo, Dora Giammarresi, and Stefano Varricchio:
Finite Automata and Non-Self-Embedding Grammars. In: *Conference on Implementation and Application of Automata (CIAA) 2002*. Lecture Notes in Computer Science, vol. 2608, pp. 47–56. Springer, 2002. DOI: 10.1007/3-540-44977-9_4.

[Aho+06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman:
Compilers: Principles, Techniques, and Tools (2nd Edition). Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[Alb85] Maris Alberts:
Space Complexity of Alternating Turing Machines. In: *Fundamentals of Computation Theory (FCT) '85*. Lecture Notes in Computer Science, vol. 199, pp. 1–7. Springer, 1985. ISBN: 3-540-15689-5. DOI: 10.1007/BFb0028785.

[AS03] Jean-Paul Allouche and Jeffrey O. Shallit:
Automatic Sequences - Theory, Applications, Generalizations. Cambridge University Press, 2003. ISBN: 978-0-521-82332-6.

[Bed+12] Zuzana Bednárová, Viliam Geffert, Carlo Mereghetti, and Beatrice Palano: The Size-Cost of Boolean Operations on Constant Height Deterministic Pushdown Automata. *Theoretical Computer Science* 449, pp. 23–36, 2012. ISSN: 03043975. DOI: 10.1016/j.tcs.2012.05.009.

[Bed+14] Zuzana Bednárová, Viliam Geffert, Carlo Mereghetti, and Beatrice Palano: Removing Nondeterminism in Constant Height Pushdown Automata. *Information and Computation* 237, pp. 257–267, 2014.

[Bed+16] Zuzana Bednárová, Viliam Geffert, Klaus Reinhardt, and Abuzer Yakaryilmaz: New Results on the Minimum Amount of Useful Space. *International Journal of Foundations of Computer Science* 27(2), pp. 259–282, 2016. DOI: 10.1142/S0129054116400098.

[Bir96] Jean-Camille Birget: Two-Way Automata and Length-Preserving Homomorphisms. *Mathematical Systems Theory* 29(3), pp. 191–226, 1996. DOI: 10.1007/BF01201276.

[BL77] Piotr Berman and Andrzej Lingas: *On the Complexity of Regular Languages in Terms of Finite Automata*. Tech. rep. 304. Polish Academy of Sciences, 1977.

[Boj+17] Mikołaj Bojańczyk, Laure Daviaud, Bruno Guillon, and Vincent Penelle: Which Classes of Origin Graphs Are Generated by Transducers. In: *International Colloquium on Automata, Languages, and Programming (ICALP) 2017*. Leibniz International Proceedings in Informatics, vol. 80, 114:1–13. 2017.

[Cho59a] Noam Chomsky: On Certain Formal Properties of Grammars. *Information and Control* 2(2), pp. 137–167, 1959. DOI: 10.1016/S0019-9958(59)90362-6.

[Cho59b] Noam Chomsky:
A Note on Phrase Structure Grammars. *Information and Control* 2(4), pp. 393–395, 1959. DOI: 10.1016/S0019-9958(59)80017-6.

[Cho62] Noam Chomsky:
Context-Free Grammars and Pushdown Storage. Tech. rep. 65. Research Laboratory of Electronics: Massachusetts Institute of Technology, 1962, pp. 187–194.

[CL15] Vincent Carnino and Sylvain Lombardy:
On Determinism and Unambiguity of Weighted Two-Way Automata. *International Journal of Foundations of Computer Science* 26(8), pp. 1127–1146, 2015. DOI: 10.1142/S0129054115400158.

[CS63] Noam Chomsky and Marcel-Paul Schützenberger:
The Algebraic Theory of Context-Free Languages. In: *Computer Programming and Formal Systems*. Studies in Logic and the Foundations of Mathematics, vol. 35. Elsevier, 1963, pp. 118–161. DOI: 10.1016/S0049-237X(08)72023-8.

[Gaj15] David Gajser:
Verifying Time Complexity of Turing Machines. *Theoretical Computer Science* 600, pp. 86–97, 2015. DOI: 10.1016/j.tcs.2015.07.028.

[Gef91] Viliam Geffert:
Nondeterministic Computations in Sublogarithmic Space and Space Constructibility. *SIAM Journal on Computing* 20(3), pp. 484–498, 1991. DOI: 10.1137/0220031.

[GGP14] Viliam Geffert, Bruno Guillon, and Giovanni Pighizzini:
Two-Way Automata Making Choices Only At the Endmarkers. *Information and Computation* 239, pp. 71–86, 2014. ISSN: 08905401. DOI: 10.1016/j.ic.2014.08.009.

[GMP03] Viliam Geffert, Carlo Mereghetti, and Giovanni Pighizzini:
Converting Two-Way Nondeterministic Unary Automata into Simpler Au-

tomata. *Theoretical Computer Science* 295(1–3), pp. 189 –203, 2003. ISSN: 0304-3975. DOI: http://dx.doi.org/10.1016/S0304-3975(02)00403-6.

[GMP07] Viliam Geffert, Carlo Mereghetti, and Giovanni Pighizzini:
Complementing Two-Way Finite Automata. *Information and Computation* 205(8), pp. 1173–1187, 2007.

[GMP10] Viliam Geffert, Carlo Mereghetti, and Beatrice Palano:
More Concise Representation of Regular Languages by Automata and Regular Expressions. *Information and Computation* 208(4), pp. 385–394, 2010.

[Gol+02] Jonathan Goldstine, Martin Kappes, Chandra M. R. Kintala, Hing Leung, Andreas Malcher, and Detlef Wotschke:
Descriptional Complexity of Machines with Limited Resources. *Journal of Universal Computer Science* 8(2), pp. 193–234, 2002. DOI: 10.3217/jucs-008-02-0193.

[GP11] Viliam Geffert and Giovanni Pighizzini:
Two-Way Unary Automata versus Logarithmic Space. *Information and Computation* 209(7), pp. 1016–1025, 2011. DOI: 10.1016/j.ic.2011.03.003.

[GP19] Bruno Guillon and Luca Prigioniero:
Linear-Time Limited Automata. *Theoretical Computer Science* 798, pp. 95–108, 2019. DOI: 10.1016/j.tcs.2019.03.037.

[GPP18] Bruno Guillon, Giovanni Pighizzini, and Luca Prigioniero:
Non-Self-Embedding Grammars, Constant-Height Pushdown Automata, and Limited Automata. In: *Conference on Implementation and Application of Automata (CIAA) 2018*. Lecture Notes in Computer Science, vol. 10977, pp. 186–197. Springer, 2018. DOI: 10.1007/978-3-319-94812-6_16.

[GR62] Seymour Ginsburg and Henry G. Rice:
Two Families of Languages Related to ALGOL. *Journal of the ACM* 9(3), pp. 350–371, 1962. DOI: 10.1145/321127.321132.

[Gui+18] Bruno Guillon, Giovanni Pighizzini, Luca Prigioniero, and Daniel Průša:
Two-Way Automata and One-Tape Machines - Read Only versus Linear Time.
In: *Developments in Language Theory (DLT) 2018*. Lecture Notes in Computer Science, vol. 11088, pp. 366–378. Springer, 2018. DOI: 10.1007/978-3-319-98654-8_30.

[Göd31] Kurt Gödel:
Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik* 38(1), pp. 173–198, 1931. DOI: 10.1007/bf01700692.

[Har67] Juris Hartmanis:
Context-Free Languages and Turing Machine Computations. In: *Mathematical Aspects of Computer Science*. Proceedings of Symposia in Applied Mathematics, vol. 19, pp. 42–51. American Mathematical Society, 1967.

[Har78] Michael A. Harrison:
Introduction to Formal Language Theory. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1978. ISBN: 0201029553.

[Hen65] Fred C. Hennie:
One-Tape, Off-Line Turing Machine Computations. *Information and Control* 8(6), pp. 553–578, 1965.

[Hib67] Thomas N. Hibbard:
A Generalization of Context-Free Determinism. *Information and Control* 11(1/2), pp. 196–238, 1967.

[HK10] Markus Holzer and Martin Kutrib:
Descriptional Complexity — An Introductory Survey. In: *Scientific Applications of Language Methods*. Imperial College Press, 2010, pp. 1–58. DOI: 10.1142/9781848165458_0001.

[HK19] Markus Holzer and Martin Kutrib:
One-Time Nondeterministic Computations. *International Journal of Foundations of Computer Science* 30(6-7), pp. 1069–1089, 2019. DOI: 10.1142/S012905411940029X.

[HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman:
Introduction to automata theory, languages, and computation - (2. ed.) Addison-Wesley-Longman, 2001. ISBN: 978-0-201-44124-6.

[HR89] Juris Hartmanis and Desh Ranjan:
Space Bounded Computations: Review and New Separation Results. In: *Mathematical Foundations of Computer Science (MFCS) '89*. Lecture Notes in Computer Science, vol. 379, pp. 49–66. Springer, 1989. DOI: 10.1007/3-540-51486-4_56.

[HS03] Juraj Hromkovic and Georg Schnitger:
Nondeterminism versus Determinism for Two-Way Finite Automata: Generalizations of Sipser's Separation. In: *International Colloquium on Automata, Languages, and Programming (ICALP) 2003*. Lecture Notes in Computer Science, vol. 2719, pp. 439–451. Springer, 2003. ISBN: 3-540-40493-7. DOI: 10.1007/3-540-45061-0_36.

[HSS17] Yo-Sub Han, Arto Salomaa, and Kai Salomaa:
Ambiguity, Nondeterminism and State Complexity of Finite Automata. *Acta Cybernetica* 23(1), pp. 141–157, 2017. DOI: 10.14232/actacyb.23.1.2017.9.

[HU69] John E. Hopcroft and Jeffrey D. Ullman:
Formal languages and their relation to automata. Addison-Wesley series in computer science and information processing. Addison-Wesley, 1969. ISBN: 0201029839.

[HU79] John E. Hopcroft and Jeffrey D. Ullman:
Introduction to Automata Theory, Languages and Computation. Addison-Wesley, 1979.

[Huf54] David A. Huffman:
The Synthesis of Sequential Switching Circuits. *Journal of the Franklin Institute* 257(4), pp. 275–303, 1954. ISSN: 0016-0032. DOI: 10.1016/0016-0032(54)90618-3.

[Kap13] Christos A. Kapoutsis:
Nondeterminism Is Essential in Small Two-Way Finite Automata with Few Reversals. *Information and Computation* 222, pp. 208–227, 2013. DOI: 10.1016/j.ic.2012.11.001.

[Kap14a] Christos A. Kapoutsis:
Predicate Characterizations in the Polynomial-Size Hierarchy. In: *Language, Life, Limits - Computability in Europe (CiE) 2014.* Lecture Notes in Computer Science, vol. 8493, pp. 234–244. Springer, 2014. DOI: 10.1007/978-3-319-08019-2_24.

[Kap14b] Christos A. Kapoutsis:
Two-Way Automata Versus Logarithmic Space. *Theory of Computing Systems* 55(2), pp. 421–447, 2014. DOI: 10.1007/s00224-013-9465-0.

[Kel84] Alica Kelemenová:
Complexity of Normal Form Grammars. *Theoretical Computer Science* 28, pp. 299–314, 1984. DOI: 10.1016/0304-3975(83)90026-9.

[KKM12] Christos A. Kapoutsis, Richard Královic, and Tobias Mömke:
Size Complexity of Rotating and Sweeping Automata. *Journal of Computer and System Sciences* 78(2), pp. 537–558, 2012. DOI: 10.1016/j.jcss.2011.06.004.

[Kle52] Stephen Cole Kleene:
Introduction to Metamathematics. North Holland, 1952.

[KP15] Christos A. Kapoutsis and Giovanni Pighizzini:
Two-Way Automata Characterizations of L/poly Versus NL. *Theory of Computing Systems* 56(4), pp. 662–685, 2015. DOI: 10.1007/s00224-014-9560-x.

[KPW18] Martin Kutrib, Giovanni Pighizzini, and Matthias Wendlandt:
Descriptional Complexity of Limited Automata. *Information and Computation* 259(2), pp. 259–276, 2018. DOI: 10.1016/j.ic.2017.09.005.

[KS19] Chris Keeler and Kai Salomaa:
Nondeterminism Growth and State Complexity. In: *Descriptional Complexity of Formal Systems (DCFS) 2019*. Lecture Notes in Computer Science, vol. 11612, pp. 210–222. Springer, 2019. DOI: 10.1007/978-3-030-23247-4_16.

[Kur64] Sige-Yuki Kuroda:
Classes of Languages and Linear-Bounded Automata. *Information and Control* 7(2), pp. 207–223, 1964. DOI: 10.1016/S0019-9958(64)90120-2.

[KW15] Martin Kutrib and Matthias Wendlandt:
On Simulation Cost of Unary Limited Automata. In: *Descriptional Complexity of Formal Systems (DCFS) 2015*. Lecture Notes in Computer Science, vol. 9118, pp. 153–164. Springer, 2015. ISBN: 978-3-319-19224-6.

[Leu05] Hing Leung:
Descriptional Complexity of NFA of Different Ambiguity. *International Journal of Foundations of Computer Science* 16(5), pp. 975–984, 2005. DOI: 10.1142/S0129054105003418.

[Mal+12] Andreas Malcher, Katja Meckel, Carlo Mereghetti, and Beatrice Palano:
Descriptional Complexity of Pushdown Store Languages. *Journal of Automata, Languages and Combinatorics* 17(2-4), pp. 225–244, 2012. DOI: 10.25596/jalc-2012-225.

[Mea55] George H. Mealy:
A Method for Synthesizing Sequential Circuits. *The Bell System Technical Journal* 34(5), pp. 1045–1079, 1955. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1955.tb03788.x.

[MF71] Albert R. Meyer and Michael J. Fischer:
Economy of Description by Automata, Grammars, and Formal Systems. In: *Switching Automata Theory (SwAT) 1971*, pp. 188–191. IEEE Computer Society, 1971.

[Moo56] Edward F. Moore:
Gedanken-Experiments on Sequential Machines. In: *Automata studies*. Princeton University Press, 1956, pp. 129–153. DOI: 10.2307/2964500.

[MP00] Carlo Mereghetti and Giovanni Pighizzini:
Two-Way Automata Simulations and Unary Languages. *Journal of Automata, Languages and Combinatorics* 5(3), pp. 287–300, 2000.

[MP43] Warren S. McCulloch and Walter Pitts:
A Logical Calculus of the Ideas Immanent in Nervous Activity. *The bulletin of mathematical biophysics* 5(4), pp. 115–133, 1943. ISSN: 1522-9602. DOI: 10.1007/BF02478259.

[MP71] Robert McNaughton and Seymour A. Papert:
Counter-Free Automata (M.I.T. Research Monograph No. 65). The MIT Press, 1971. ISBN: 0262130769.

[Ner58] Anil Nerode:
Linear Automaton Transformations. *Proceedings of the American Mathematical Society* 9(4), pp. 541–544, 1958. ISSN: 00029939, 10886826.

[Okh12] Alexander Okhotin:
Non-erasing Variants of the Chomsky-Schützenberger Theorem. In: *Developments in Language Theory (DLT) 2012*. Lecture Notes in Computer Science, vol. 7410. Springer, 2012, pp. 121–129. DOI: 10.1007/978-3-642-31653-1_12.

[ORW85] William F. Ogden, Rockford J. Ross, and Karl Winklmann:
An "Interchange Lemma" for Context-Free Languages. *SIAM Journal on Computing* 14(2), pp. 410–415, 1985. DOI: 10.1137/0214031.

[Pig09a] Giovanni Pighizzini:
Deterministic Pushdown Automata and Unary Languages. *International Journal of Foundations of Computer Science* 20(4), pp. 629–645, 2009.

[Pig09b] Giovanni Pighizzini:
Nondeterministic One-Tape Off-Line Turing Machines. *Journal of Automata, Languages and Combinatorics* 14(1), pp. 107–124, 2009.

[Pig13] Giovanni Pighizzini:
Two-Way Finite Automata: Old and Recent Results. *Fundamenta Informaticae* 126(2-3), pp. 225–246, 2013. DOI: 10.3233/FI-2013-879.

[Pig15] Giovanni Pighizzini:
Guest Column: One-Tape Turing Machine Variants and Language Recognition. *SIGACT News* 46(3), pp. 37–55, 2015. DOI: 10.1145/2818936.2818947.

[Pig16] Giovanni Pighizzini:
Strongly Limited Automata. *Fundamenta Informaticae* 148(3-4), pp. 369–392, 2016.

[Pig19] Giovanni Pighizzini:
Limited Automata: Properties, Complexity and Variants. In: *Descriptional Complexity of Formal Systems (DCFS) 2019*. Lecture Notes in Computer Science, vol. 11612, pp. 57–73. Springer, 2019. DOI: 10.1007/978-3-030-23247-4_4.

[PP14] Giovanni Pighizzini and Andrea Pisoni:
Limited Automata and Regular Languages. *International Journal of Foundations of Computer Science* 25(7), pp. 897–916, 2014. DOI: 10.1142/S0129054114400140.

[PP15] Giovanni Pighizzini and Andrea Pisoni:
Limited Automata and Context-Free Languages. *Fundamenta Informaticae* 136(1-2), pp. 157–176, 2015. DOI: 10.3233/FI-2015-1148.

[PP17] Giovanni Pighizzini and Luca Prigioniero:
Non-Self-Embedding Grammars and Descriptional Complexity. In: *Non-Classical Models of Automata and Applications (NCMA) 2017*, pp. 197–209. 2017.

[PP19a] Giovanni Pighizzini and Luca Prigioniero:
Limited Automata and Unary Languages. *Information and Computation* 266, pp. 60–74, 2019. DOI: 10.1016/j.ic.2019.01.002.

[PP19b] Giovanni Pighizzini and Luca Prigioniero:
Pushdown Automata and Constant Height: Decidability and Bounds. In: *Descriptional Complexity of Formal Systems (DCFS) 2019*. Lecture Notes in Computer Science, vol. 11612, pp. 260–271. Springer, 2019. DOI: 10.1007/978-3-030-23247-4_20.

[Prů14] Daniel Průša:
Weight-Reducing Hennie Machines and Their Descriptional Complexity. In: *Language and Automata Theory and Applications (LATA) 2014*. Lecture Notes in Computer Science, vol. 8370, pp. 553–564. 2014.

[PSW02] Giovanni Pighizzini, Jeffrey O. Shallit, and Ming-wei Wang:
Unary Context-Free Grammars and Pushdown Automata, Descriptional Complexity and Auxiliary Space Lower Bounds. *Journal of Computer and System Sciences* 65(2), pp. 393–414, 2002.

[Rad62] Tibor Radó:
On Non-Computable Functions. *Bell System Technical Journal* 41(3), pp. 877–884, 1962. ISSN: 0018-8646.

[RS59] Michael O. Rabin and Dana Scott:
Finite Automata and Their Decision Problems. *IBM Journal of Research and Development* 3(2), pp. 114–125, 1959. DOI: 10.1147/rd.32.0114.

[Sav70] Walter J. Savitch:
Relationships between Nondeterministic and Deterministic Tape Complexi-

ties. *Journal of Computer and System Sciences* 4(2), pp. 177–192, 1970. DOI: 10.1016/S0022-0000(70)80006-X.

[Sha08] Jeffrey O. Shallit:
A Second Course in Formal Languages and Automata Theory. Cambridge University Press, 2008. ISBN: 978-0-521-86572-2.

[She59] John C. Shepherdson:
The Reduction of Two-Way Automata to One-Way Automata. *IBM Journal of Research and Development* 3(2), pp. 198–200, 1959. DOI: 10.1147/rd.32.0198.

[Sip80] Michael Sipser:
Halting Space-Bounded Computations. *Theoretical Computer Science* 10(3), pp. 335–338, 1980. ISSN: 0304-3975.

[Sloa] Neil J. A. Sloane:
The On-Line Encyclopedia of Integer Sequences. http://oeis.org/A007814.

[Slob] Neil J. A. Sloane:
The On-Line Encyclopedia of Integer Sequences. http://oeis.org/A001511.

[SS78] William J. Sakoda and Michael Sipser:
Nondeterminism and the Size of Two Way Finite Automata. In: *Symposium on Theory of Computing (SToC) 1978*, pp. 275–286. ACM, 1978. DOI: 10.1145/800133.804357.

[Tur36] Alan M. Turing:
On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* s2-42(1), pp. 230–265, 1936. ISSN: 0024-6115. DOI: 10.1112/plms/s2-42.1.230.

[TYL10] Kohtaro Tadaki, Tomoyuki Yamakami, and Jack C. H. Lin:
Theory of One-Tape Linear-Time Turing Machines. *Theoretical Computer Science* 411(1), pp. 22–43, 2010. DOI: 10.1016/j.tcs.2009.08.031.

[WW86] Klaus W. Wagner and Gerd Wechsung:
Computational Complexity. Dordrecht: D. Reidel Publishing Company, 1986.