Limited Automata and Unary Languages[☆]

Giovanni Pighizzini[*], Luca Prigioniero

Dipartimento di Informatica, Università degli Studi di Milano, Italy

Abstract

Limited automata are one-tape Turing machines that can rewrite the contents of tape cells only in the first d visits, for a fixed d. When $d = 1$ these models characterize regular languages.

We show an exponential gap between the size of limited automata accepting unary languages and the size of equivalent finite automata. 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 G it is possible to obtain an equivalent 1-limited automaton whose description has a size that is polynomial in the size of G.

For alphabets of cardinality at least 2, for each grammar G generating a context-free language L, it is possible to obtain a 1-limited automaton whose description has polynomial size in that of G and whose accepted language L' is Parikh-equivalent to L.

Keywords: unary languages, limited automata, context-free grammars, Parikh equivalence

1. Introduction

The investigation of computational models and of their computational power is a classical topic in computer science. For instance, characterizations of the classes in the Chomsky hierarchy by different types of computational devices are well-known. In particular, the class of context-free languages is characterized in terms of pushdown automata. A less known characterization of this class was obtained in 1967 by Hibbard, in terms of Turing machines with rewriting restrictions, called *limited automata* [1]. For each integer $d \geq 0$, a d-limited automaton is a one-tape Turing machine that can rewrite the contents of each tape cell *only in the first d visits*. For each $d \geq 2$, the class of languages accepted by these devices coincides with the class of context-free languages, while for $d = 1$ only regular languages are accepted [1, 2].

More recently, limited automata have been investigated from the descriptional complexity point of view, by studying the relationships between the sizes of their descriptions and those of other equivalent formal systems. Pighizzini and Pisoni proved that each 1-limited automaton M 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 M is deterministic [3]. The same authors showed how to transform each given 2-limited automaton M into an equivalent pushdown automaton, having a description of exponential size with respect to the description of M. Even for this simulation, the cost cannot be reduced in the worst case. On the other hand, the converse simulation is polynomial in size [4]. Kutrib, Pighizzini, and Wendlandt proved that the cost of the simulation of d-limited automata by pushdown automata is still bounded by an exponential function even when $d > 2$ [5]. A subclass of 2-limited automata, which still characterizes context-free languages but whose members are polynomially related in size with pushdown automata, has been investigated in [6].

[☆]Extended version of a paper presented at *DLT 2017 - Developments in Language Theory*, Liège, Aug 7–11, 2017 [Lecture Notes in Computer Science, vol. 10396 Springer, 2017, pp. 308–319.]

[*]Corresponding author

Email addresses: pighizzini@di.unimi.it (Giovanni Pighizzini), prigioniero@di.unimi.it (Luca Prigioniero)

In all above-mentioned results, lower bounds 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 [7] 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 [3]. 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 [4].

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 [8].

In this paper 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 will 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 paper, we consider unary context-free grammars. The cost of the conversion of these grammars into finite automata has been investigated by proving exponential gaps [9]. Here, we study the conversion of unary context-free grammars into limited automata. With the help of a result presented by Okhotin [10], we prove that each unary context-free grammar G can be converted into an equivalent 1-limited automaton whose description has a size that is polynomial in the size of G.

This result is finally generalized to all context-free grammars, namely by removing the unary restriction, considering the notion of *Parikh equivalence*. We remind the reader that two strings over a same alphabet Σ are Parikh-equivalent when they are equal up to a permutation of their symbols or, equivalently, for each letter $a \in \Sigma$, the number of occurrences of a in the two strings is the same. Two languages L' and L'' are Parikh-equivalent if for each x in L' there is a string y in L'' which is Parikh-equivalent to x and vice versa. It is well-known that each context-free language has a Parikh-equivalent regular language *(Parikh's Theorem)* [11]. The size costs of the conversion of context-free grammars into nondeterministic and deterministic finite automata accepting Parikh-equivalent languages have been studied by Lavado, Pighizzini, and Seki [12]. Here, we prove that each context-free grammar G can be converted into a 1-limited automaton whose description has a size that is polynomial in the size of G and whose accepted language is Parikh-equivalent to the language generated by G. This also gives an alternative proof of Parikh's Theorem.

2. Preliminaries

In this section we recall some basic definitions which are used in the paper. Given a set S, $\#S$ denotes its cardinality and 2^S the family of all of its subsets. Given an alphabet Σ and a string $w \in \Sigma^*$, let us denote by $|w|$ the length of w and by ε the empty string.

We assume the reader familiar with notions from formal languages and automata theory, in particular with the fundamental variants of finite automata (1DFAs, 1NFAs, 2DFAs, 2NFAs, for short, where 1 and 2 mean *one-way* and *two-way*, respectively, and D and N mean *deterministic* and *nondeterministic*, respectively) and with *context-free grammars* (CFGs, for short). For further details see, e.g., [13].

Given an integer $d \geq 0$, a *d-limited automaton* (d-LA, for short) is a tuple $A = (Q, \Sigma, \Gamma, \delta, q_I, F)$, where Q is a finite set of states, Σ is a finite input alphabet, Γ is a finite *working alphabet* such that $\Sigma \cup \{\rhd, \lhd\} \subseteq \Gamma$, $\rhd, \lhd \notin \Sigma$ are two special symbols, called the *left* and the *right end-markers*, respectively, $\delta : Q \times \Gamma \to 2^{Q \times \Gamma \times \{-1,+1\}}$ is the transition function. At the beginning of the computation, the input 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 automaton is on the cell 1 and the state of the finite control is the *initial state* q_I. In one move, according to δ and to the current state, 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 to one position forward or backward. In particular, $(q, X, m) \in \delta(p, a)$ means that when the automaton in the state p is scanning a cell containing the symbol a, it can enter the state q, rewrite the cell contents by X, and move the head to *left*, if $m = -1$, or to *right*, if $m = +1$. 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 $X \in \{\triangleright, \triangleleft\}$ or $a \in \{\triangleright, \triangleleft\}$, 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 below.

Further technical details (which are not necessary in this paper) can be found in [4].

An automaton A is said to be *limited* if it is d-limited for some $d \geq 0$. A accepts an input w if and only if there is a computation path which starts from the initial state q_I with the input tape containing w surrounded by the two end-markers and the head on the first input cell, and which ends in a *final state* $q \in F$ 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. The language accepted by A is denoted by $L(A)$. A is said to be *deterministic* (d-DLA, for short) whenever $\#\delta(q, a) \leq 1$, for any $q \in Q$ and $a \in \Gamma$.

In this paper we are interested in comparing the sizes of the descriptions of devices and formal systems. As customary, to measure the size of a finite automaton we consider the cardinality of the state set. For a context-free grammar we count the total number of symbols which are used to write down its productions.

For d-limited automata, the size depends on the number q of states and on the cardinality m of the working alphabet. In fact, given these two parameters, the possible number of transitions is bounded by $2q^2m^2$. Hence, if q and m are polynomial with respect to given parameters, also the size of the d-LA is polynomial.

The costs of the simulations of 1-LAs by finite automata have been investigated in [3], proving the following result:

Theorem 1. *Let M be an n-state* 1-LA*. Then 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 M is deterministic then an equivalent* 1DFA *with no more than $n \cdot (n+1)^n$ states can be obtained.*

Using witness languages over a binary alphabet, it has been shown that in the worst case exponentially and doubly exponentially many states are necessary to simulate an n-state 1-LA by a 1NFA and a 1DFA, respectively. Furthermore, also the simulation of deterministic 1-LAs by 1DFAs requires, in the worst case, exponentially many states [3].

Each context-free grammar $G = (V, \Sigma, P, S)$ can be converted into a grammar G' which does not contain ε-productions and generates the language $L(G) - \{\varepsilon\}$. Furthermore, the grammar G' can be obtained of size linear in the size of G [14]. For this reason, in the paper *we will only consider grammars without ε-productions*.

A context-free grammar G is in *double Greibach normal form* if its productions have the forms $A \to bC_1 \cdots C_k d$ and $A \to a$, with $a, b, d \in \Sigma$, $k \geq 0$, and $A, C_1, \ldots, C_k$ are variables [15]. Each context-free grammar G (not generating the empty word) can be converted into an equivalent one in double Greibach normal form. Furthermore, the size of the description of the resulting grammar is polynomial in the size of the description of G [16, 17].

A language L is *local* if there exist sets $\mathcal{A} \subseteq \Sigma \times \Sigma$, $\mathcal{I} \subseteq \Sigma$, and $\mathcal{F} \subseteq \Sigma$ such that $w \in L$ 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 [18]. It is not difficult to verify that local languages are a subclass of regular languages.

The *Parikh map* $\psi : \Sigma^* \to \mathbb{N}^m$, $m = \#\Sigma$, associates with any word $w \in \Sigma^*$ the vector which counts the occurrences of each letter of Σ in w. The vector $\psi(w)$ is also called *Parikh image* of w. One can naturally generalize this map for a language $L \subseteq \Sigma^*$ as

$$\psi(L) = \{\psi(w) \mid w \in L\}.$$

The set $\psi(L)$ is called the *Parikh image* of L. Two languages $L, L' \subseteq \Sigma^*$ are said to be *Parikh-equivalent* to each other, in symbols $L \equiv_\pi L'$, if and only if $\psi(L) = \psi(L')$.

Parikh equivalence can be defined not only between languages but among languages, grammars, and finite automata by referring, in the last two cases, to the defined languages. By *Parikh's Theorem*, each context-free language is Parikh-equivalent to a regular language [11].

3. 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 $n > 1$, the language $L_n = \{a^{2^n}\}$, which requires $2^n + 1$ states to be accepted by a 1NFA, can be accepted by a 1-DLA of size $O(n)$. Let us proceed by steps. In order to illustrate the construction, first it is useful to discuss how L_n can be accepted by a linear bounded automaton M_n (i.e., a Turing machine that can use as storage the tape which initially contains the input, by rewriting its cells an unbounded number of times).

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 M_n discovers that the number of a's on the tape was odd then M_n rejects.

iii. After the last iteration, M_n accepts if only one a is left on the tape.

It is possible to modify M_n, without any increasing of the number of the states, by introducing a different kind of writing at step i.: at the first iteration, the machine uses the symbol 0 instead of X for rewriting, at the second one it uses the symbol 1, and so on. After the n-th iteration, only one cell of the tape should contain the symbol a. In this case, M_n writes the symbol n on such a cell and accepts; otherwise, 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 M_n will be formally studied in Lemma 1.)

Considering the last extension of M_n, we are now going to introduce a 1-LA N_n accepting the language L_n, based on the guessing of the final tape contents of M_n.

In the first phase, 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 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 $O(n)$ accepting L_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 M_n when it accepts the input:

Lemma 1. *At the end of the computation of 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 rewritten 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* [19].[1]

From Lemma 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 $j > 0, k \geq 0$:

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

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.

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 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 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 1-DLA 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 $s = k_1k_2\cdots k_j$, 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$(k_1k_2\cdots k_j) = (i_1, i_2, \ldots, i_r)$, $j, r > 0$, 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 $1 > 0$),

[1] In [20], 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 [21].

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 2. *For $j > 0$, if* $\text{BIS}(\sigma_1\sigma_2\cdots\sigma_j) = (i_1, i_2, \ldots, i_r)$ *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, $\text{BIS}(\sigma_1\sigma_2\cdots\sigma_j) = (k)$.
- 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 (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, $\text{BIS}(\sigma_1\sigma_2\cdots\sigma_j)$ can be obtained by appending k at the end of $\text{BIS}(\sigma_1\sigma_2\cdots\sigma_{j'})$, namely, if $\text{BIS}(\sigma_1\sigma_2\cdots\sigma_{j'}) = (i_1, \ldots, i_{r'})$, $r' = r - 1$, then $\text{BIS}(\sigma_1\sigma_2\cdots\sigma_j) = (i_1, \ldots, i_{r'}, i_r)$, $i_r = k$.

  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 2, we now prove a property which will be crucial in the construction of a 1-DLA accepting the language L_n.

Lemma 3. *For $j > 0$, σ_j is the smallest integer greater than or equal to 0 not occurring in* $\text{BIS}(\sigma_1\sigma_2\cdots\sigma_{j-1})$.

Proof. In the case $j = 1$, $\text{BIS}(\sigma_1\sigma_2\cdots\sigma_{j-1})$ 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 $j-1$.[2] Considering Lemma 2, applied to $j - 1$, we conclude that σ_j is the smallest integer not occurring in $\text{BIS}(\sigma_1\sigma_2\cdots\sigma_{j-1})$. □

We are going to define a 1-DLA $A_n = (Q, \Sigma, \Gamma, \delta, q_I, F)$ accepting the language $L_n = \{a^{2^n}\}$. The automaton 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, A_n writes the symbol σ_j of the binary carry sequence that, according to Lemma 3, can be computed as the smallest nonnegative integer missing in $\text{BIS}(\sigma_1\sigma_2\cdots\sigma_{j-1})$. 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 A_n implements the procedure summarized in Algorithm 1 — note that, for ease of presentation, the algorithm assumes that the machine starts the computation with the head on the left end-marker — and it is defined as follows: $Q = \{q_I, q_F, q_1, \ldots, q_n, p_1, \ldots, p_{n-1}\}$, $\Sigma = \{a\}$, $\Gamma = \{0, \ldots, n\}$, 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. $\delta(p_{n-1}, n-1) = (q_n, n-1, +1)$

[2] 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.

Algorithm 1: Recognition of the language L_n

```
1  start with the head on the left end-marker
2  while symbol under the head ∉ {n, ◁} do
3  |  move the head to the right
4  |  write 0
5  |  j ← 0
6  |  repeat
7  |  |  while symbol under the head ≤ j and ≠ ▷ do
8  |  |  |  move the head to the left
9  |  |  j ← j + 1
10 |  until symbol under the head ≠ j
11 |  repeat
12 |  |  move the head to the right
13 |  until symbol under the head = a
14 |  write j
15 if symbol under the head = n then
16 |  move the head to the right
17 |  if symbol under the head = ◁ then Accept
18 Reject
```

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

vii. $\delta(q_i, a) = (q_I, i, +1)$, 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 A_n has $2n + 1$ states, which is linear in the parameter n.

The machine starts in the initial state q_I. 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 3 and 4). 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 $\textsc{bis}(\sigma_1, \ldots, \sigma_p)$. This is done with transitions from ii. to v. — lines from 6 to 10. After that, A_n moves its head to the right until the first a is reached (transitions vi. — lines from 11 to 13) and writes the symbol j (transitions vii. — line 14). 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 15 to 17). In the latter case, A_n stops and rejects (undefined transition).

Hence we conclude that the language L_n is accepted by a 1-DLA 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 1-DLAs and 2NFAs there is the same gap. This gives the main result of this section:

Theorem 2. *For each integer $n > 1$ there exists a unary language K_n such that K_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 2^n states.*

Proof. With a few minor changes, the above presented automaton A_n can accept $K_n = \{a^{2^n}\}^*$. From Theorem 9 in [22], each 2NFA requires at least 2^n states to accept the same language. □

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

Theorem 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 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 $qm^{1/2} \geq cn^{1/2}$.

4. Unary Grammars versus Limited Automata

In Section 3 we proved an exponential gap between unary 1-LAS and finite automata. A similar gap was obtained between unary CFGS and finite automata [9]. Hence, it is natural to study the size relationships between unary CFGS and 1-LAS. Here, we prove that each context-free grammar G specifying a unary language can be converted into an equivalent 1-LA M of polynomial size. More precisely, the sizes of the set of states and of the working alphabet of M are polynomial with respect to the size of G.

Let us start by presenting some notions and preliminary results.

Definition 1. *A* bracket alphabet Ω_b *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 D_{Ω_b} *over Ω_b is the set of all sequences of balanced brackets from Ω_b.*

An extended bracket alphabet Ω *is a nonempty finite set which is the union of two, possibly empty, sets Ω_b and Ω_n, where Ω_b, if not empty, is a bracket alphabet, and Ω_n is a set of* neutral *symbols. The* extended Dyck language $\widehat{D}_\Omega$ *over Ω is the set of all the strings that can be obtained by arbitrarily inserting symbols from Ω_n in strings of D_{Ω_b}. Given an integer $d > 0$, the* extended Dyck language with nesting depth bounded by d *over Ω, denoted as $\widehat{D}^{(d)}_\Omega$, is the subset of $\widehat{D}_\Omega$ consisting of all strings where the nesting depth of brackets is at most d.*

Example 1. Let $\Omega_b = \{$ (, [,) ,] $\}$, $\Omega_n = \{$ | $\}$, and $\Omega = \Omega_b \cup \Omega_n$. Then ([[ ]]) [ ] $\in D_{\Omega_b} \subset \widehat{D}_\Omega$, | (| [[ ] |]) [|] $\in \widehat{D}^{(3)}_\Omega \setminus \widehat{D}^{(2)}_\Omega$. □

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

Proof. We can define a 2DFA M which verifies the membership of its input w to $\widehat{D}^{(d)}_\Omega$ by using a counter c. During a first scan M checks whether or not the brackets are correctly nested, *regardless* their types. This is done as follows. Starting with 0 in c, 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 M rejects. 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, M verifies that corresponding left and right brackets are of the same type. To this aim, starting from the left end-marker, M moves its head to the right, to locate a left bracket. When it is found, 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, M verifies

the matching with the one saved in the control. If this is not the case, then M stops and rejects. Otherwise, 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, 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, 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. □

The following nonerasing variant of the Chomsky-Schützenberger representation theorem for context-free languages, proved by Okhotin [10], is crucial to obtain our main result:

Theorem 4. *A language $L \subseteq \Sigma^*$ is context-free if and only if there exist an extended bracket alphabet Ω_L, a regular language $R_L \subseteq \Omega_L^*$, and a letter-to-letter homomorphism $h : \Omega_L \to \Sigma$ such that $L = h(\widehat{D}_{\Omega_L} \cap R_L)$.*

In [6], it was observed that the language R_L of Theorem 4 is local and the size of the alphabet Ω is polynomial with respect to the size of a context-free grammar G generating L. This was used to prove that each context-free grammar 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 G. In the following, when L is specified by a context-free grammar G, i.e., $L = L(G)$, we will write Ω_G and R_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 4, we prove the following result:

Theorem 5. *Let $L \subseteq \{a\}^*$ be a unary regular language and $G = (V, \{a\}, P, S)$ be a context-free grammar of size s generating it. Then, there exist an extended bracket alphabet Ω_G and a regular language $\widehat{R}_G \subseteq \Omega_G^*$ such that $L = h(\widehat{D}^{(\#V)}_{\Omega_G} \cap \widehat{R}_G)$, where:*

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

Furthermore, the size of Ω_G is polynomial in the size s of the grammar G and the language $\widehat{R}_G$ 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 Section 5.

Given a context-free grammar $G = (V, \{a\}, P, S)$ specifying a unary language L, we first obtain the representation in Theorem 4. According to Theorem 5.2 in [6], the size of the alphabet Ω_G is polynomial with respect to the size of the description of G. Each pair of brackets in Ω_G represents the root of a derivation tree of G, which starts from a certain variable of G and produces a terminal string.

If a sequence $w \in \Omega_G^*$ 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 $\widehat{D}_{\Omega_G}$, we can obtain an "equivalent" string of the same length in $\widehat{D}^{(\#V)}_{\Omega_G}$.

The regular language R_G should be modified accordingly. While in the representation in Theorem 4, the first and the last symbol of a string $w \in \widehat{D}_{\Omega_G} \cap R_G$ 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_G, can be verified by a 2DFA with a polynomial number of states. The details will be given in the proof of Theorem 7. □

Notice that if we omit the state bound for the 2DFA accepting $\widehat{R}_G$, the statement of Theorem 5 becomes trivial: by taking $\Omega_G = \{a\}$ where a is a neutral symbol, $\widehat{R}_G = L$, and $h(a) = a$, we obtain $\widehat{D}^{(\#V)}_{\Omega_G} = \{a\}^*$ and hence $\widehat{D}^{(\#V)}_{\Omega_G} \cap \widehat{R}_G = L = h(L)$.

Using Theorem 5, we now prove the main result of this section:

Theorem 6. *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 $G = (V, \{a\}, P, S)$ be the given grammar, $L \subseteq \{a\}^*$ be the unary language generated by it, Ω_G be the extended bracket alphabet, and $\widehat{R}_G$ be the regular language obtained from G according to Theorem 5.

We define a 1-LA M which works as follows:

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

According to Lemma 4, step 2 can be done by simulating a 2DFA with $O(\#\Omega_G \cdot \#V)$ states, hence a number polynomial in s. Furthermore, by Theorem 5, also step 3 can be performed by simulating a 2DFA with a number of states polynomial in s. Hence M has a size that is polynomial in s. □

We point out that from Theorem 6 and the exponential gap from unary CFGS to 1NFAS proved in [9], we could derive an exponential gap from unary *nondeterministic* 1-LAS to 1NFAS. In Section 3 we proved that the gap remains exponential if we restrict to unary *deterministic* 1-LAS and consider the simulation by 2NFAS.

5. Proof of Theorem 5 and More

The primary purpose of this section is to present a detailed proof of Theorem 5. Actually, we are going to prove a stronger result which will allow us to obtain a generalization of Theorem 6 by showing, for each context-free grammar, the existence of a polynomial-size 1-LA Parikh-equivalent to it.

Theorem 7. *Let $L \subseteq \Sigma^*$ be a regular language and $G = (V, \Sigma, P, S)$ be a context-free grammar of size s generating it. Then, there exist an extended bracket alphabet Ω_G and a regular language $\widehat{R}_G \subseteq \Omega_G^*$ such that L is Parikh-equivalent to $h(\widehat{D}^{(\#V)}_{\Omega_G} \cap \widehat{R}_G)$, where:*

- *$\widehat{D}^{(\#V)}_{\Omega_G}$ is the extended Dyck language over Ω_G with nesting depth bounded by $\#V$,*
- *h is a letter-to-letter homomorphism from Ω_G to Σ.*

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

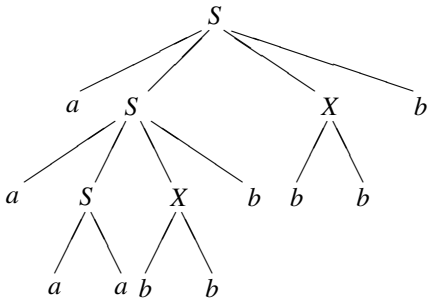

Figure 1: A derivation tree of the string $w = aaaabbbbbb$ in the grammar G of Example 2.

Since two unary languages are Parikh-equivalent exactly when they are equal, Theorem 5 derives from Theorem 7, by taking $\Sigma = \{a\}$.

Let us start by presenting the proof of Theorem 7. First of all, let us recall a classical argument used to prove the pumping lemma for context-free languages (a similar approach has been used in the unary case in [9], and in the case of bounded languages in [23]).

Let $G = (V, \Sigma, P, S)$ be a context-free grammar without ε-productions. If T is a tree representing a derivation in G of $\beta \in (V \cup \Sigma)^*$ from a variable A, then we write $T : A \stackrel{\star}{\Rightarrow} \beta$. We also denote by $\nu(T)$ the set of variables which are labels of nodes in T.

Consider a derivation tree $T : S \stackrel{\star}{\Rightarrow} z$, such that in a path from the root to a leaf two different nodes n_1 and n_2 have the same label $B \in V$. Suppose that n_1 is closer to the root than n_2. Let uBy be the string generated by the tree obtained by removing from T all the descendants of n_1, vBx be the string generated by the tree T'' which is obtained by the subtree rooted at n_1 by removing all the descendants of n_2, and w be the string generated by the tree rooted at n_2. Then $z = uvwxy$ with $vx \neq \varepsilon$. By replacing in T the subtree rooted at n_1 with the subtree rooted at n_2, we obtain a new tree $T' : S \stackrel{\star}{\Rightarrow} uwy$, with $|uwy| < |z|$. Furthermore, $T'' : B \stackrel{+}{\Rightarrow} vBx$, $B \in \nu(T')$, and $\nu(T) = \nu(T') \cup \nu(T'')$. Notice that concatenation of the strings generated by T' and T'', after eliminating the symbol B, i.e., the string $uwyvx$, is Parikh-equivalent to the string $z = uvwxy$ generated by the original tree T.

Let us call any tree that derives a string of terminals from S a *terminal tree*. Furthermore, a *gap tree on a variable B* (*B-gap tree*, for short) is any tree whose leaves are labeled by terminal symbols, with the exception of one leaf which is labeled with the same variable B as the root. The above T and T' are examples of terminal trees, while T'' is a gap tree.

Example 2. To illustrate our construction we elaborate an example from [10], by considering the grammar $G = (V, \Sigma, P, S)$, with $V = \{S, X\}$, $\Sigma = \{a, b\}$ and P the set containing the productions

$$\begin{array}{lcl} S & \rightarrow & aSXb \mid aa \\ X & \rightarrow & bb \end{array}$$

which generates the language $\{a^{n+2}b^{3n} \mid n \geq 0\}$. In Figure 1, a derivation tree T for the string $w = aaaabbbbbb$ is depicted. In Figure 2, a terminal tree T' and an S-gap tree T'' obtained from T are represented. Notice that the concatenation of the strings generated by T' and T'' (by dropping the leaf labeled with S in the gap tree) is Parikh equivalent to w. □

By summarizing the previous discussion, each terminal tree T whose depth exceeds $\#V$ can be decomposed into a terminal tree T' and a gap tree T'' on a variable B, such that $B \in \nu(T')$, $\nu(T) = \nu(T') \cup \nu(T'')$, and the concatenation of the *terminal* symbols generated by T' and T'' is Parikh-equivalent to the string generated by T. The trees T' and T'' obtained from such a decomposition may still have depth greater than $\#V$. However, as we will discuss soon, in such a case we can further decompose them in order to reduce the depth. To this aim, we observe that, in a similar way, each gap tree U of depth greater than $\#V$ on a variable B can be decomposed in trees U' and U'', where U' is a gap tree

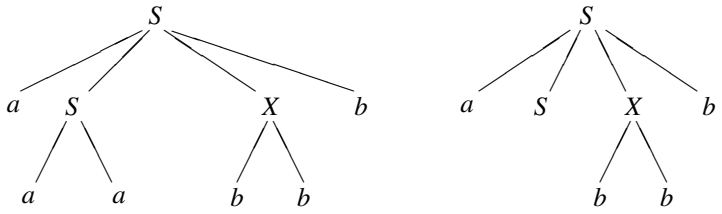

Figure 2: A terminal tree T' and an S-gap tree T'' obtained from the tree in Figure 1.

on B, U'' is a gap tree on a variable C, with $C \in \nu(U')$, $\nu(U) = \nu(U') \cup \nu(U'')$, and the concatenation of the *terminal* symbols generated by U' and U'' is Parikh-equivalent to the sequence of *terminal* symbols generated by U.

Hence, given a terminal tree T of depth greater than $\#V$, first we decompose it into a terminal tree T' and a gap tree T'', as above explained. Each of the resulting trees which have depth greater than $\#V$ is further decomposed and replaced by the two resulting tree. This decomposition process is repeated up to have only trees of depth at most $\#V$. In such a way, each terminal tree T can be reduced to a terminal tree T_0 followed by a sequence of gap trees $T_1, T_2, \ldots, T_m$ on variables $B_1, \ldots, B_m$, respectively, such that $B_i \in \nu(T_0) \cup \cdots \cup \nu(T_{i-1})$, $i = 1, \ldots, m$, $\nu(T) = \nu(T_0) \cup \nu(T_1) \cup \cdots \cup \nu(T_m)$, the string generated by T is Parikh-equivalent to the string obtained by concatenating all terminal symbols generated by $T_0, T_1, \ldots, T_m$, and all the trees $T_0, T_1, \ldots, T_m$ have depth at most $\#V$.

Moreover, given a terminal tree T', a variable $B \in \nu(T')$ and a gap tree T'' on B, we can suitably "insert" T'' in T' in order to obtain a terminal tree T, with $\nu(T) = \nu(T') \cup \nu(T'')$, which derives a string which is Parikh-equivalent to the string generated by T' concatenated with the terminal symbols generated by T''. As for the decomposition, even the insertion of gap trees can be iterated. This allows us to conclude:

Lemma 5. *Let $L' \subseteq \Sigma^*$ be the set of all strings z such that there exist a terminal tree T_0 and $m \geq 0$ gap trees $T_1, T_2, \ldots, T_m$ on variables $B_1, B_2, \ldots, B_m$, respectively, such that all the following holds:*

(i) the depth of each tree T_i is bounded by $\#V$, for $i = 0, \ldots, m$,

(ii) $B_i \in \nu(T_0) \cup \cdots \cup \nu(T_{i-1})$, for $i = 1, \ldots, m$,

(iii) $T_0 : S \overset{\star}{\Rightarrow} z_0$, for some $z_0 \in \Sigma^+$,

(iv) $T_i : B_i \overset{\star}{\Rightarrow} v_i B_i x_i$, for some $v_i, x_i \in \Sigma^*$ with $v_i x_i \neq \varepsilon$, for $i = 1, \ldots, m$,

(v) $z = z_0 v_1 x_1 v_2 x_2 \cdots v_m x_m$.

Then $L' \equiv_\pi L(G)$.

In the construction given to prove Theorem 4, each string $w \in \widehat{D}_{\Omega_G} \cap R_G$ linearly encodes a terminal tree $T : S \overset{\star}{\Rightarrow} z$, with $|z| = |w|$. In particular, any factor v of w delimited by a matching pair of left and right brackets encodes a subtree $T' : B \overset{\star}{\Rightarrow} u$ of T, with $|u| = |v|$. While the purpose of $\widehat{D}_{\Omega_G}$ is only to guarantee the correct matching of brackets, the language R_G allows to select, among the sequences in $\widehat{D}_{\Omega_G}$, only those ones which encode terminal trees of G. To this purpose, R_G is defined in such a way that the first and the last symbol of each string $w \in R_G$ should be a matching pair of brackets representing the start of a derivation from S, while each 2-letter factor in w represents one step in the derivation process.

To prove Theorem 7, we will adapt the techniques which have been used in [10] to prove Theorem 4. However, a terminal tree T will be replaced by a bounded depth terminal tree and a sequence of bounded depth gap trees, satisfying the conditions of Lemma 5. Indeed, according to such a lemma, there exists a language L' which is Parikh equivalent to $L(G)$ such that the membership of a string to L' is witnessed by a terminal tree T_0 and a sequence of gap trees $T_1, \ldots, T_m$, $m \geq 0$, satisfying the conditions of Lemma 5. The tree T_0 derives a prefix z_0 of z. Each tree T_i, $i = 1, \ldots, m$, derives from a variable B_i the string $v_i B_i x_i$, where only the terminal part $v_i x_i$ is a factor of z, while the "gap variable" B_i is used to verify that the gap tree corresponds to a valid decomposition of a tree in G (condition (ii)).

Because of condition (i), $\widehat{D}_{\Omega_G}$ can be restricted to $\widehat{D}^{(\#V)}_{\Omega_G}$, which is regular. However, the language R_G should be replaced by a larger language $\widehat{R}_G$ in such a way that among all strings $w \in D^{(\#V)}_{\Omega_G}$, exactly those encoding sequences of trees satisfying condition (ii) are allowed.

We now present the details. First, let us summarize the main points in the proof of Theorem 4 [10], together with their complexity aspects [6].

- First of all, the grammar G can be assumed to be in *double Greibach normal form*. This is useful in order to obtain a letter-to-letter homomorphism. As mentioned in Section 2, the conversion into this form can polynomially increase the size of the description. With a further polynomial increase of the size, we can also suppose that for each production $A \to bC_1 \cdots C_k d$, $i \neq j$ implies $C_i \neq C_j$, for $i, j = 1, \ldots, k$.

- The extended bracket alphabet Ω_G consists of brackets of the form $(^{\Xi}_{A\to bC_1\cdots C_k d}$ and $)^{\Xi}_{A\to bC_1\cdots C_k d}$, for each production $A \to bC_1 \cdots C_k d$, and neutral symbols of the form $|^{\Xi}_{A\to a}$, for each production $A \to a$, where $A \to bC_1 \cdots C_k d$ or $A \to a$ represent the *current production* and, in both cases, Ξ is either a production (the *previous production*) or the symbol "$-$" (in the case the current production is the *starting production* which is applied at the outer level of the derivation tree). Hence the size of Ω_G is quadratic in the number of the productions of G, which means that it is polynomial in the size of the description of G.

- The regular language R_G is defined in order to allow only sequences of brackets which encode derivations of G. Intuitively, each factor delimited by the matching pair of brackets $(^{\Xi}_{A\to\alpha}$, $)^{\Xi}_{A\to\alpha}$ in a string in $\widehat{D}_{\Omega_G} \cap R_G$ represents a derivation in G which starts by using the production $A \to \alpha$ and produces a terminal string (a clarifying picture from [10] is given in Figure 3).

  To this aim, it is enough to define the language R_G in terms of local conditions. More precisely, the set of 2-letter factors which are allowed in a string is defined to check whether the productions are applied in a consistent order, and it is the following:

  - For each production $A \to bC_1 \cdots C_k d$, $k \geq 1$, and $\Xi \in P \cup \{-\}$, and all productions $C_1 \to \gamma_1, C_2 \to \gamma_2, \ldots, C_k \to \gamma_k$, indices $i = 2, \ldots, k$:

    $(^{\Xi}_{A\to bC_1\cdots C_k d}\,(^{A\to bC_1\cdots C_k d}_{C_1\to\gamma_1}$ if $|\gamma_1| > 1$,

    $(^{\Xi}_{A\to bC_1\cdots C_k d}\,|^{A\to bC_1\cdots C_k d}_{C_1\to a}$ if $\gamma_1 = a$,

    $)^{A\to bC_1\cdots C_k d}_{C_{i-1}\to\gamma_{i-1}}\,(^{A\to bC_1\cdots C_k d}_{C_i\to\gamma_i}$ if $|\gamma_{i-1}| > 1$ and $|\gamma_i| > 1$,

    $|^{A\to bC_1\cdots C_k d}_{C_{i-1}\to a}\,(^{A\to bC_1\cdots C_k d}_{C_i\to\gamma_i}$ if $\gamma_{i-1} = a$ and $|\gamma_i| > 1$,

    $|^{A\to bC_1\cdots C_k d}_{C_{i-1}\to a}\,|^{A\to bC_1\cdots C_k d}_{C_i\to a'}$ if $\gamma_{i-1} = a$ and $\gamma_i = a'$,

    $)^{A\to bC_1\cdots C_k d}_{C_{i-1}\to\gamma_{i-1}}\,|^{A\to bC_1\cdots C_k d}_{C_i\to a}$ if $|\gamma_{i-1}| > 1$ and $\gamma_i = a$,

    $)^{A\to bC_1\cdots C_k d}_{C_k\to\gamma_k}\,)^{\Xi}_{A\to bC_1\cdots C_k d}$ if $|\gamma_k| > 1$,

    $|^{A\to bC_1\cdots C_k d}_{C_k\to a}\,)^{\Xi}_{A\to bC_1\cdots C_k d}$ if $\gamma_k = a$.

  - For each production $A \to bd$ and $\Xi \in P \cup \{-\}$:

    $(^{\Xi}_{A\to bd})^{\Xi}_{A\to bd}$

  - The symbols which are allowed at the beginning and at the end of strings in R_G are:

    $(^{-}_{S\to\sigma}$ and $)^{-}_{S\to\sigma}$, respectively, where $S \to \sigma \in P$ with $\sigma \neq a$,

    $|^{-}_{S\to a}$, if $S \to a \in P$.

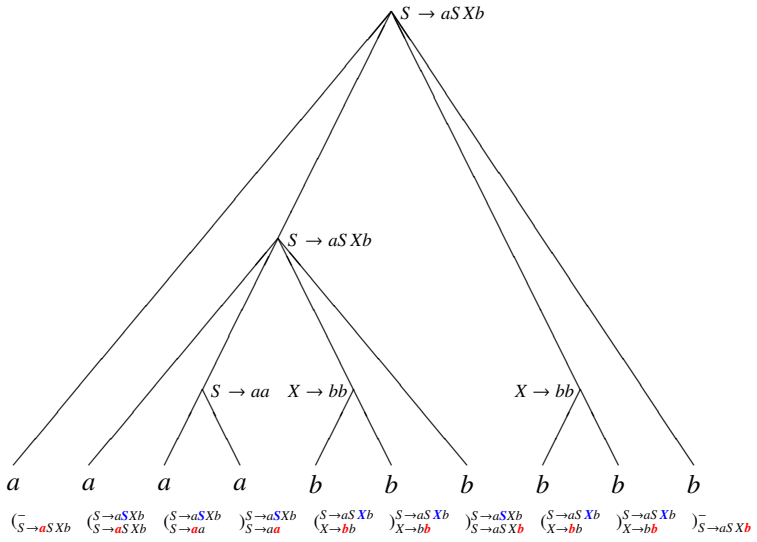

Figure 3: The derivation tree of Figure 1, with the corresponding bracket string [10].

Observing the list of allowed 2-symbol factors, we can conclude that a neutral symbol can appear at the beginning or at the end of a string $x \in R_G$ only if $|x| = 1$. Furthermore, if $|x| \geq 2$ and $x \in \widehat{D}_G \cap R_G$ then the first and the last symbol of x form a matching pair.

- The homomorphism $h : \Omega_G \to \Sigma$ is defined as follows:

$$h\left((^{\Xi}_{A\to bC_1\cdots C_k d}\right) = b\,, \qquad h\left()^{\Xi}_{A\to bC_1\cdots C_k d}\right) = d\,, \qquad h\left(|^{\Xi}_{A\to a}\right) = a\,,$$

for productions $A \to bC_1 \cdots C_k d$, $A \to a$, and $\Xi \in P \cup \{-\}$.

While we keep the same extended bracket alphabet Ω_G and the same homomorphism $h : \Omega_G \to \Sigma$, we have to replace the language R_G by a larger language $\widehat{R}_G$ that we now describe. Each string $w \in \widehat{D}^{(\#V)}_{\Omega_G} \cap \widehat{R}_G$ should be factorized as $w_0 w_1 \cdots w_m$, $m \geq 0$, where the string w_0 encodes a terminal tree T_0, while the strings $w_1, \ldots, w_m$ encode gap trees $T_1, \ldots, T_m$ on variables $B_1, \ldots, B_m$, respectively, satisfying the conditions (i) and (ii) of Lemma 5. To this aim, the language $\widehat{R}_G$ is defined by several requirements.

We notice that, since the grammar G is in double Greibach normal form, each gap tree generates at least two terminal symbols, i.e., the production used at the top level of a B-gap tree is of the form $B \to \beta$, with $\beta \in bV^+d$.

The first group of requirements is the following:

- The set $\mathcal{A}$ of allowed 2-letter factors in strings of $\widehat{R}_G$ contains all the above listed 2-letter factors which are allowed in strings of R_G, plus the factor $)^{-}_{A\to\alpha}\,(^{-}_{B\to\beta}$ for all productions $A \to \alpha, B \to \beta$, with $|\alpha|, |\beta| > 1$. When the grammar contains the production $S \to a$, also the 2-letter factors $|^{-}_{S\to a}\,(^{-}_{B\to\beta}$ are allowed, for all productions $B \to \beta$, with $|\beta| > 1$. All these 2-letter factors include the last symbol encoding one tree T_{i-1} and the first symbol encoding the next tree T_i.[3] Notice that at the moment we are ignoring the problem of representing the leaf labeled by B is any B-gap tree, which will be discussed later.

- The symbols which are allowed at the beginning of a string are, as in R_G, $(^{-}_{S\to\alpha}$, for $S \to \alpha \in P$, with $|\alpha| > 1$, and $|^{-}_{S\to a}$, when $S \to a \in P$.

- The symbols which are allowed at the end of a string are $)^{-}_{A\to\alpha}$, for all productions $A \to \alpha$, with $|\alpha| > 1$, and $|^{-}_{S\to a}$, when $S \to a \in P$.

[3] In the case of gap trees, since the variable labeling the root must appear on one leaf, the first production $B \to \beta$ should generate at least one variable, thus implying $|\beta| > 2$. This will be implicitly verified by checking the existence of a B-gap factor, as explained later on.

With these conditions, a string of $w \in \widehat{D}^{(\#V)}_{\Omega_G} \cap \widehat{R}_G$ would encode a sequence of *terminal* trees $T_0, T_1, \ldots, T_m$, $m \geq 0$, where each tree is delimited by a pair of matching brackets $(^{-}_{A\to\alpha},)^{-}_{A\to\alpha}$ (for a production $A \to \alpha$ with $|\alpha| > 1$). When the grammar contains the production $S \to a$, T_0 could also be the tree which is encoded by $|^{-}_{S\to a}$.

However, in each sequence of trees $T_0, T_1, \ldots, T_m$ we are considering, only the first tree T_0 should be a terminal tree, while the remaining trees should be gap trees. Hence, each one of them should derive a string which contains, besides terminal symbols, one occurrence of the same variable labeling the root. To solve this problem, we require that the encoding of each gap tree contains exactly one 2-letter factor which is obtained by removing the leaf labeled with the same variable B at the root from a 3-letter factor containing B. To this aim, we define the set $\mathcal{G}_B$ of *B-gap factors*. These factors are similar to those in the set $\mathcal{A}$, but they are obtained by skipping the leaf labeled B. In the encoding of a gap tree on a variable B, it is required that the list of all 2-letter factors contains one factor from $\mathcal{G}_B$, while all the other factors are from the set $\mathcal{A}$.[4] (For an example, see Figure 4.) This is the list of elements in $\mathcal{G}_B$, for $B \in V$:

- For each production $A \to bBd$ and $\Xi \in P \cup \{-\}$:

  $(^{\Xi}_{A\to bBd})^{\Xi}_{A\to bBd}$

- For each production $A \to bC_1 \cdots C_k d$, $k > 1$, and $\Xi \in P \cup \{-\}$, and all productions $C_1 \to \gamma_1, C_2 \to \gamma_2, \ldots, C_k \to \gamma_k$, indices $i = 2, \ldots, k$:

$$
\begin{array}{ll}
(^{\Xi}_{A\to bC_1\cdots C_k d}\,(^{A\to bC_1\cdots C_k d}_{C_2\to\gamma_2} & \text{if } C_1 = B \text{ and } |\gamma_2| > 1,\\
(^{\Xi}_{A\to bC_1\cdots C_k d}\,|^{A\to bC_1\cdots C_k d}_{C_2\to a} & \text{if } C_1 = B \text{ and } \gamma_2 = a,\\
)^{A\to bC_1\cdots C_k d}_{C_{i-1}\to\gamma_{i-1}}\,(^{A\to bC_1\cdots C_k d}_{C_{i+1}\to\gamma_i} & \text{if } C_i = B,\ |\gamma_{i-1}| > 1 \text{ and } |\gamma_{i+1}| > 1,\\
|^{A\to bC_1\cdots C_k d}_{C_{i-1}\to a}\,(^{A\to bC_1\cdots C_k d}_{C_{i+1}\to\gamma_i} & \text{if } C_i = B,\ \gamma_{i-1} = a \text{ and } |\gamma_{i+1}| > 1,\\
|^{A\to bC_1\cdots C_k d}_{C_{i-1}\to a}\,|^{A\to bC_1\cdots C_k d}_{C_{i+1}\to a'} & \text{if } C_i = B,\ \gamma_{i-1} = a \text{ and } \gamma_{i+1} = a',\\
)^{A\to bC_1\cdots C_k d}_{C_{i-1}\to\gamma_{i-1}}\,|^{A\to bC_1\cdots C_k d}_{C_{i+1}\to a} & \text{if } C_i = B,\ |\gamma_{i-1}| > 1 \text{ and } \gamma_{i+1} = a,\\
)^{A\to bC_1\cdots C_k d}_{C_{k-1}\to\gamma_{k-1}}\,)^{\Xi}_{A\to bC_1\cdots C_k d} & \text{if } C_k = B \text{ and } |\gamma_{k-1}| > 1,\\
|^{A\to bC_1\cdots C_k d}_{C_{k-1}\to a}\,)^{\Xi}_{A\to bC_1\cdots C_k d} & \text{if } C_k = B \text{ and } \gamma_{k-1} = a.
\end{array}
$$

Notice that the variable B corresponding to the root of a B-gap tree can be obtained from the pair of matching brackets delimiting the encoding of the tree, namely it should appear on the left hand side in the production defining the two "starting" brackets $(^{-}_{B\to\alpha}$ and $)^{-}_{B\to\alpha}$. We also observe that the gap trees $T_1, \ldots, T_m$ should satisfy the condition (ii) of Lemma 5. This happens when, for $i = 1, \ldots, m$, the variable B_i labeling the root of the gap tree T_i, already occurred in some of the symbols in the prefix to the left of $(^{-}_{B_i\to\alpha}$, $i = 1, \ldots, m$. Only strings satisfying such a condition can belong to the language $\widehat{R}_G$.

We now describe a 2DFA M accepting the language $\widehat{R}_G$. If the first input symbol is of the form $(^{-}_{S\to\sigma}$, for a production $S \to \sigma$, with $|\sigma| > 1$, then M inspects the input from left to right, by checking that each 2-letter factor belongs to the set $\mathcal{A}$, up to reach a cell containing $)^{-}_{S\to\sigma}$, which completes the encoding of the tree T_0. In this phase, M needs only to remember the last input symbol. Now, M continues to inspect the remaining input cells. A cell c containing a bracket of the form $(^{-}_{B\to\alpha}$ indicates the beginning of a gap tree T on variable B and it should immediately be preceded by a cell containing a bracket of the form $)^{-}_{A\to\alpha'}$. Before starting the inspection of the input part starting from the cell c, where the encoding of T is expected, M checks whether the condition (ii) of Lemma 5 is satisfied. To this aim M moves its head to the left, to search the first cell containing a bracket of the form $)^{\Xi}_{B\to\beta}$ or $|^{\Xi}_{B\to\beta}$ for some $\Xi \in P \cup \{-\}$, $B \to \beta \in P$, i.e., the last symbol where the variable B occurred. If the search is unsuccessful then M

[4] Note that the sets $\mathcal{A}$ and $\mathcal{G}_B$ are not necessarily disjoint.

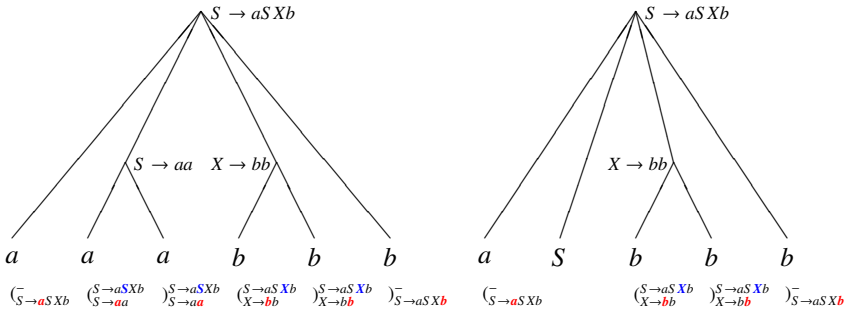

Figure 4: The terminal tree T' and the gap tree T'' of Figure 2, with the corresponding bracket strings. Note the gap factor $(^{-}_{S\to aSXb}(^{S\to aSXb}_{X\to bb}$, corresponding to the 3-letter factor aSb in the string $aSbbb$ derived by T''.

stops and rejects. Otherwise, M moves the head again to the cell c which can be located by moving to the right, as the first cell containing a symbol of the form $(^{-}_{B\to\alpha}$. This part can be implemented just remembering the variable B, so it uses $O(\#V)$ states.

Now, M has to verify that in the input factor from the cell c up to the first cell containing a symbol of the form $(^{-}_{B'\to\alpha'}$, for some production $B' \to \alpha'$, or before the right end-marker, there is one 2-letter factor belonging to the set $\mathcal{G}_B$, while all the remaining 2-letter factors belong to $\mathcal{A}$. Since, as already observed, the sets $\mathcal{G}_B$ and $\mathcal{A}$ are not necessarily disjoint, to do this M has to check that either there is exactly one 2-letter factor belonging to $\mathcal{G}_B \setminus \mathcal{A}$, while the remaining 2-letter factors belong to $\mathcal{A}$, or all 2-letter factors belong to $\mathcal{A}$ and at least one of them also belongs to $\mathcal{G}_B$. This can be done by inspecting from left to right with $O(\#\Omega_G)$ states.

M explores all the input with the same technique. When the right end-marker is reached, M needs also to verify that the last input cell corresponds to the end of a gap tree or of a terminal tree. So, it has to check whether the last input symbol was of the form $)^{-}_{B\to\alpha}$, for some production $B \to \alpha$. If the outcome of this check is positive, then M stops and accepts.

In the case the grammar contains the production $S \to a$, the first input symbol could be $|^{-}_{S\to a}$. If it is the only symbol on the tape, then M accepts. Otherwise, M can apply the above described process to inspect the remaining symbols on the tape. Notice that in this case the tape could encode a sequence of trees $T_0, T_1, \ldots, T_m$, where T_0 derives a from S, and T_1 is a gap tree on S.

This completes the proof of Theorem 7. □

6. Context-Free Grammars versus Parikh-Equivalent Limited Automata

Each context-free grammar of size s can be converted into an equivalent 2-limited automaton, whose number of states is polynomial in s [4]. Since 1-limited automata accept only regular languages, we cannot convert context-free grammars into them. However, given a context-free grammar G we can find a 1-limited automaton accepting a language which is Parikh-equivalent to the language generated by G. In this section we study the size cost of such a conversion.

We mention that each context-free grammar of size s can be converted into a Parikh-equivalent deterministic finite automaton with a number of states exponential in a polynomial in s. Furthermore, the exponential gap cannot be reduced even converting the grammar into a nondeterministic finite automaton [12]. Extending Theorem 6, we prove that the conversion into 1-limited automata is polynomial:

Theorem 8. *Each context-free grammar of size s can be converted into a Parikh-equivalent* 1-LA *having a size that is polynomial in s.*

Proof. Given a context-free grammar $G = (V, \Sigma, P, S)$ generating a language $L \subseteq \Sigma^*$, we consider the extended bracket alphabet Ω_G, the regular language $\widehat{R}_G$, and the homomorphism h obtained from G according to Theorem 7. Then, the language $L' = h(\widehat{D}^{(\#V)}_{\Omega_G} \cap \widehat{R}_G)$ is Parikh-equivalent to L.

We define a 1-LA M accepting L' and having a size polynomial in s as in the proof of Theorem 6, with the only difference that in step 1, to rewrite the contents of each input cell, M nondeterministically chooses a symbol in $h^{-1}(a)$,

where $a \in \Sigma$ is the input symbol in the cell. In this way, on input $z \in \Sigma^*$, the string $w \in \Omega_G^*$ which is written on the tape at the end of step 1 satisfies $h(w) = z$. □

7. Conclusion

Using languages defined over a binary alphabet, exponential size gaps were proved for the conversion of 1-LAs into 2NFAs and of 1-DLAs into 1DFAs [3], 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 in [3] 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 4 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 [4]. The cost remains exponential when we convert d-LAs into CFGs, for each $d > 2$ [5].

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.

Our construction in Section 5 is derived from the non-erasing variant of the Chomsky-Schützenberger Theorem obtained by Okhotin [10]. Another non-erasing variant of the same theorem has been recently proved by Crespi Reghizzi and San Pietro [24]. In this version, the bracket alphabet only depends on the terminal alphabet of the grammar, more precisely its size is polynomial in the size of the terminal alphabet, while in the version we used the size is polynomial in the number of productions. Such a reduction is payed in the definition of the regular language, which is no more local (or, equivalently, 2-local), but k-local (i.e., it is testable using a "window" of k symbols), for some constant k, which is logarithmically related to the grammar complexity (details are discussed in the extended version [25]).

One could try to prove Theorem 7, by considering this variant of the Chomsky-Schützenberger Theorem. However, our techniques do not seem directly applicable, since our construction uses the information on the grammar which is encoded in the bracket alphabet (indeed, when the encoding of a B-gap tree is found, the previous part of the bracket string is scanned to verify that the variable B already occurred), while in this variant the bracket alphabet is not related to the grammar.

Acknowledgments

The authors would like to thank the anonymous referees for the careful revision work. Their valuable remarks and recommendations were very helpful in order to improve the quality of the paper.

References

[1] T. N. Hibbard, A generalization of context-free determinism, Information and Control 11 (1/2) (1967) 196–238.

[2] K. W. Wagner, G. Wechsung, Computational Complexity, D. Reidel Publishing Company, Dordrecht, 1986.

[3] G. Pighizzini, A. Pisoni, Limited automata and regular languages, Int. J. Found. Comput. Sci. 25 (7) (2014) 897–916. URL http://dx.doi.org/10.1142/S0129054114400140

[4] G. Pighizzini, A. Pisoni, Limited automata and context-free languages, Fundam. Inform. 136 (1-2) (2015) 157–176. URL http://dx.doi.org/10.3233/FI-2015-1148

[5] M. Kutrib, G. Pighizzini, M. Wendlandt, Descriptional complexity of limited automata, Inf. Comput. 259 (2) (2018) 259–276. URL https://doi.org/10.1016/j.ic.2017.09.005

[6] G. Pighizzini, Strongly limited automata, Fundam. Inform. 148 (3-4) (2016) 369–392. URL http://dx.doi.org/10.3233/FI-2016-1439

[7] S. Ginsburg, H. G. Rice, Two families of languages related to ALGOL, J. ACM 9 (3) (1962) 350–371. URL http://doi.acm.org/10.1145/321127.321132

[8] M. Kutrib, M. Wendlandt, On simulation cost of unary limited automata, in: J. Shallit, A. Okhotin (Eds.), DCFS 2015, Vol. 9118 of Lecture Notes in Computer Science, Springer, 2015, pp. 153–164.
URL http://dx.doi.org/10.1007/978-3-319-19225-3_13

[9] G. Pighizzini, J. Shallit, M. Wang, Unary context-free grammars and pushdown automata, descriptional complexity and auxiliary space lower bounds, J. Computer and System Sciences 65 (2) (2002) 393–414.

[10] A. Okhotin, Non-erasing variants of the Chomsky-Schützenberger theorem, in: H. Yen, O. H. Ibarra (Eds.), DLT 2012, Vol. 7410 of Lecture Notes in Computer Science, Springer, 2012, pp. 121–129.
URL http://dx.doi.org/10.1007/978-3-642-31653-1_12

[11] R. Parikh, On context-free languages, J. ACM 13 (4) (1966) 570–581.
URL http://doi.acm.org/10.1145/321356.321364

[12] G. J. Lavado, G. Pighizzini, S. Seki, Converting nondeterministic automata and context-free grammars into Parikh equivalent one-way and two-way deterministic automata, Inf. Comput. 228 (2013) 1–15.
URL https://doi.org/10.1016/j.ic.2013.06.003

[13] J. E. Hopcroft, J. D. Ullman, Introduction to Automata Theory, Languages and Computation, Addison-Wesley, 1979.

[14] J. Gruska, A note on ε-rules in context-free grammars, Kybernetika 11 (1) (1975) 26–31.
URL http://www.kybernetika.cz/content/1975/1/26

[15] D. J. Rosenkrantz, Matrix equations and normal forms for context-free grammars, J. ACM 14 (3) (1967) 501–507.

[16] J. Engelfriet, An elementary proof of double Greibach normal form, Inf. Process. Lett. 44 (6) (1992) 291–293.

[17] R. Yoshinaka, An elementary proof of a generalization of double Greibach normal form, Inf. Process. Lett. 109 (10) (2009) 490–492.
URL http://dx.doi.org/10.1016/j.ipl.2009.01.015

[18] R. McNaughton, S. A. Papert, Counter-Free Automata (M.I.T. Research Monograph No. 65), The MIT Press, 1971.

[19] N. J. A. Sloane, The On-Line Encyclopedia of Integer Sequences, http://oeis.org/A007814.

[20] J. Allouche, J. O. Shallit, Automatic Sequences - Theory, Applications, Generalizations, Cambridge University Press, 2003.
URL http://www.cambridge.org/gb/knowledge/isbn/item1170556/

[21] N. J. A. Sloane, The On-Line Encyclopedia of Integer Sequences, http://oeis.org/A001511.

[22] C. Mereghetti, G. Pighizzini, Two-way automata simulations and unary languages, Journal of Automata, Languages and Combinatorics 5 (3) (2000) 287–300.

[23] A. Malcher, G. Pighizzini, Descriptional complexity of bounded context-free languages, Inf. Comput. 227 (2013) 1–20.
URL https://doi.org/10.1016/j.ic.2013.03.008

[24] S. Crespi Reghizzi, P. San Pietro, The missing case in Chomsky-Schützenberger theorem, in: A. Dediu, J. Janousek, C. Martín-Vide, B. Truthe (Eds.), Language and Automata Theory and Applications - 10th International Conference, LATA 2016, Prague, Czech Republic, March 14-18, 2016, Proceedings, Vol. 9618 of Lecture Notes in Computer Science, Springer, 2016, pp. 345–358.
URL https://doi.org/10.1007/978-3-319-30000-9_27

[25] S. Crespi Reghizzi, P. San Pietro, Non-erasing Chomsky-Schützenberger theorem with grammar-independent alphabet, CoRR abs/1805.04003. arXiv:1805.04003.
URL http://arxiv.org/abs/1805.04003