Charles University
Faculty of Mathematics and Physics

Mgr. Daniel Průša

Two-dimensional Languages

Doctoral Thesis

Supervisor: Martin Plátek, CSc.

Prague, 2004

Acknowledgements

The results presented in this thesis were achieved during the years 1998–2004. Many of them were achieved within the framework of grant projects. They include Grant-No. 201/02/1456 (The Grant Agency of the Czech Republic) and Grant-No. 300/2002/A-INF/MFF (The Grant Agency of Charles University, Prague).

I would like to gratefully thank my supervisor Martin Plátek, CSc. for his valuable support he provided me during writing the thesis.

Prague, November 2004 Daniel Průša

Contents

Chapter 1

Introduction

This thesis presents results on the theory of two-dimensional languages. The theory studies generalizations of formal languages to two dimensions. These generalizations can be done in many different ways. Automata working over a two-dimensional tape were firstly introduced by M. Blum and C. Hewitt in 1967. Since then, several formal models recognizing or generating two-dimensional objects have been proposed in the literature. All these approaches were initially motivated by problems arising in the framework of pattern recognition and image processing. Two-dimensional patterns also appear in studies concerning cellular automata and some other models of parallel computing.

The most common two-dimensional object is a *picture* which is a rectangular array of symbols taken from a finite alphabet. We restrict ourselves to the study of languages build from such objects. These languages are called *picture languages*.

1.1 Studied Two-dimensional Formal Models

We give the informal description of the models studied in this thesis.

Two-dimensional Turing machine

A Turing machine provided by a two-dimensional tape (which is a two-dimensional array of fields infinite in both directions) with the capability to move the head in four directions – left, right, up and down – is called a two-dimensional Turing machine. Such a machine can be used to recognize picture languages. In the initial configuration, an input picture is stored in the tape. The head scans typically the upper-left corner. The fields of the tape not containing a symbol of the input are filled by the *background* symbol # (it is also known as the *blank* symbol). The computation over the input is an analogy to computations of the model with one-dimensional tape.

The two-dimensional Turing machine can be considered as the most general two-dimensional recognizing device. The class of languages recognizable by two-dimensional Turing machines is an analogy to 0 languages of the Chomsky hierarchy (recursively enumerable languages). The origin of the model dates to late sixties of the twentieth century, when the basics of the theory were formed.

Two-dimensional finite-state automaton

We say that a two-dimensional Turing machine is *bounded* if the machine does not move the head outside the area of the input. To be more precise, the head is allowed to leave the area in a step but whenever this situation occurs, it does not rewrite the scanned background symbol and it has to be moved back in the next step.

Each bounded two-dimensional Turing machine that never rewrites any symbol of the tape is called a two-dimensional finite-state automaton. It is also known as a four-way automaton. Similarly as in the case of two-dimensional Turing machines, two-dimensional finite-state automata were proposed in beginnings of the theory of two-dimensional languages and, since then, their properties have been widely studied. The different topology of pictures (comparing to strings) has an impact on results that differ to results known from the one-dimensional theory. It can be demonstrated even on the two-dimensional finite-state automata, which, in fact, can be considered as one of the simplest recognizing devices. For example, non-deterministic two-dimensional finite-state automata are more powerful than deterministic two-dimensional finite-state automata.

Since two-dimensional finite-state automata are the natural extension of one-dimensional two-way automata, the formed class of picture languages recognizable by them is the natural candidate for the generalization of the class of regular languages.

Two-dimensional forgetting automaton

The one-dimensional forgetting automaton is a bounded Turing machine that uses a doubly linked list of fields rather than the usual tape. The automaton can erase the content of a scanned field by the special erase symbol or it can completely delete the scanned field from the list. Except these two operations, no additional rewriting of symbols in fields are allowed. One-dimensional forgetting automata were studied by P. Jancar, F. Mraz and M. Platek, e.g. in [7], where the taxonomy of forgetting automata is given, or in [8], where forgetting automata are used to characterize context-free languages.

The two-dimensional forgetting automaton is a bounded two-dimensional Turing machine that can again erase the scanned field, but it cannot delete it. It would be problematic to define this operation, since the deletion of a field from an array breaks the two-dimensional topology. These automata were studied by P. Foltyn [1] and P. Jiricka [9].

Two-dimensional on-line tessellation automaton

The two-dimensional on-line tessellation automaton was introduced by K. Inoue and A. Nakamura [5] in 1977. It is a kind of bounded two-dimensional cellular automaton. In comparison with the cellular automata, computations are performed in a restricted way – cells do not make transitions at every time-step, but rather a "transition wave" passes once diagonally across them. Each cell changes its state depending on two neighbors – the top one and the left one. The result of a computation is determined by the state the bottom-right cell finishes in.

D. Giammarresi and A. Restivo present the class of languages recognizable by this model as the ground level class of the two-dimensional theory ([2]), prior to languages recognizable by two-dimensional finite-state automata. They argue that the class proposed by them fulfills more natural requirements on such a generalization. Moreover, it is possible to use several formalisms to define the class: except tessellation automata, they include tiling systems or monadic second order logic, thus the definition is robust as in the case of regular languages.

Two-dimensional grammars with productions in context-free form

The basic variant of two-dimensional grammars with productions in context-free form was introduced by M. I. Schlesinger and V. Hlavac [22] and also by O. Matz [13]. Schlesinger and Hlavac present the grammars as a tool that can be used in the area of syntactic methods for pattern recognition. Productions of these grammars are of the following types:

$$1)\quad N \to a \qquad 2)\quad N \to A \qquad 3)\quad N \to \begin{array}{cc} A & B \end{array} \qquad 4)\quad N \to \begin{array}{c} A \\ B \end{array}$$

where A, B, N are non-terminals and a is a terminal. Pictures generated by a non-terminal are defined using recurrent rules:

- For each production of type 1), N generates a.
- For each production of type 2), if A generates P, then N generates P as well.
- For each production of type 3) (resp. 4)), if A generates P_1 and B generates P_2, where the numbers of rows (resp. columns) of P_1 and P_2 equal, then the concatenation of P_1 and P_2, where P_2 is placed after (resp. bellow) P_1, is generated by N.

In [9], P. Jiricka works with an extended form of productions. Right-hand sides consist of a general matrix of terminals and non-terminals.

All the mentioned authors call the grammars *two-dimensional context-free grammars*. We will see that many properties of the raised class are not analogous to the properties of one-dimensional context-free languages. That is the reason why we prefer the term *grammars with productions in context-free form*.

The following table shows abbreviations we use for the models.

Abbreviation	Recognizing device, resp. generative system
TM	two-dimensional Turing machine
FSA	two-dimensional finite-state automaton
OTA	two-dimensional on-line tessellation automaton
FA	two-dimensional forgetting automaton
CFPG	grammar with productions in context-free form

Furthermore, for each listed computational model, to denote the deterministic variant of the model we use the correspondent abbreviation prefixed by D

(i.e., we write DTM, $DFSA$, $DOTA$ and DFA). The class of languages recognizable, resp. generated by a model M is denoted by $L(M)$ (e.g. $L(FSA)$).

The chosen abbreviations do not reflect two-dimensionality of the models. In fact, TM, FSA and FA match the commonly used abbreviation for one-dimensional models. However, this fact should not lead to any confusions. We will refer one-dimensional models rarely and whenever we do it, we will emphasize it explicitly.

1.2 Goals

In the presented work, we focus mainly on two areas.

- Firstly, we study relationships among the two-dimensional formal models listed before.
- Secondly, we also contribute to the question of possibilities to generalize the Chomsky hierarchy to two dimensions.

As for the first goal, we are interested in the comparison of generative, resp. recognitive power of models, mutual simulations among them, questions related to time complexity. If possible, we would also like to develop general techniques or ideas of proof (e.g. the technique allowing to prove that a language is not included in $L(CFPG)$).

When speaking about a generalization, we have on mind the class of picture languages that fulfills some natural expectations. Let us assume to look for a generalization of regular languages. Then, as a primary requirement, we expect that, in some sense, the generalization includes regular languages. To be more precise, we mean if the two-dimensional generalization is restricted to languages containing pictures formed only of one row (resp. column), we get exactly the class of regular languages. Furthermore, we would like the mechanism defining languages in the generalized class to resemble mechanisms used to define regular languages (as finite state automata, regular grammars, etc.). In addition, we expect that the generalization inherits as many as possible properties (like closure properties) of the one-dimensional class. And finally, time complexity of recognition of languages in the class should also be important. In the case of regular languages we would expect that it is possible to decide the question of membership in time linear in number of fields of an input picture.

We have already noted that the class of languages recognizable by two-dimensional finite-state automata is the natural candidate for the generalization of regular languages. There are also some proposals of classes of picture languages generated by mechanisms resembling context-free grammars, however the properties of these classes do not fulfill the natural expectations.

We study the question whether the class of languages generated by grammars with productions in context-free form is a good generalization of one-dimensional context-free languages or not. In the mentioned literature regarding the grammars, there are only a few basic results on properties of the class. We would like to extend these results to have basis to make a conclusion. We will investigate the following topics:

- The comparison of $L(CFPG)$ and $L(FSA)$.

- The comparison of $L(CFPG)$ to the class of languages generated by the basic variant of the grammars.
- Closure properties of $L(CFPG)$.
- Searching for a recognizing device equivalent to the grammars. We would like to propose a device based on the idea of forgetting.
- The comparison to other proposals of the generalization of context-free languages.

1.3 Achieved Results

The main results that have been achieved categorized by models are as follows:

FSA: We give a characterization of two-dimensional finite-state automata working on inputs over one-symbol alphabets. We show that $L(FSA)$ is not closed under projection into such alphabets.

We also investigate possibilities how to simulate the automata by deterministic bounded two-dimensional Turing machines. This result is connected to work in [10], where the simulation of FSA by a deterministic FA is described. The main goal is to prove the existence of such a simulation there. In our case, we rather focus on time complexity of the simulation.

OTA: We show a connection between the automata and one-dimensional cellular automata. Under some circumstances, tessellation automata can simulate cellular automata. As a consequence, it gives us a generic way how to design tessellation automata recognizing NP-complete problems that are recognizable by one-dimensional cellular automata. This possibility speaks against the suggestion to take the class as the ground level of the two-dimensional hierarchy. Results regarding the simulation were published in [17].

FA: In [9], the technique allowing to store some information on the tape preserving the possibility to reconstruct the original content has been described. Using the technique, we show that forgetting automata can simulate tessellation automata as well as grammars with productions in context-free form (for every $CFPG$ G, there is a FA accepting the language generated by G).

TM: Questions related to time complexity of the recognition of pictures are studied in general. For convenience, we define the class of picture languages recognizable by two-dimensional non-deterministic Turing machines in polynomial time and show a relationship to the known one-dimensional variant of the class.

We also deal with lower bounds on time complexity in two dimensions. Our results are based on the generalization of so called crossing sequences and the technique related to them. It comes from [4]. Moreover, we show that a two-dimensional tape is an advantage – some one-dimensional languages can be recognized substantially faster by a Turing machine, when using the two-dimensional tape instead of the one-dimensional tape. These results have roots

in author's diploma thesis ([19]), where models of parallel Turing machines working over the two-dimensional tape are studied. Related results were published in [14] and [15].

CFPG: We show that $L(CFPG)$ has many properties that do not conform the natural requirements on a generalization of context-free languages. These properties include:

- $L(CFPG)$ is incomparable with $L(FSA)$.
- There is no analogy to the Chomsky normal form of productions. The generative power of the grammars is dependent on size of right-hand sides of productions.
- The emptiness problem is not decidable even for languages over a one-symbol alphabet.

Next results on the class include closure properties, a kind of pumping lemma or comparisons to other proposals of the generalization of context-free languages. We have already mentioned the result regarding a relationship between grammars in context-free form and two-dimensional forgetting automata – each language generated by a grammar with productions in context-free form can be recognized by a two-dimensional forgetting automaton. On the other hand, the two-dimensional forgetting automata are stronger, since they can simulate two-dimensional finite-state automata. We did not succeed in finding a suitable restriction on FA's to get a recognizing device for the class $L(CFGP)$. Attempts based on automata provided with a pushdown store were not successful too.

Some of the results we have listed above were published in [16] and [18].

1.4 Notation

In the thesis, we denote the set of natural numbers by $\mathbb{N}$, the set of integers by $\mathbb{Z}$. Next, $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$.

$\mathrm{lin} : \mathbb{N} \to \mathbb{N}$ is a function such that $\forall n \in \mathbb{N} : \ \mathrm{lin}(n) = n$.

For two functions $f, g : \mathbb{N} \to \mathbb{N}$, we write

- $f = O(g)$ iff there are $n_0, k \in \mathbb{N}$ such that $\forall n \in \mathbb{N}\ n \geq n_0 \Rightarrow f(n) \leq k \cdot g(n)$
- $f = \Omega(g)$ iff there are $n_0, k \in \mathbb{N}$ such that $\forall n \in \mathbb{N}\ n \geq n_0 \Rightarrow k \cdot f(n) \geq g(n)$
- $f = o(g)$ iff for any $k \in \mathbb{N}$ there is $n_0 \in \mathbb{N}$ such that $f(n_0) \geq k \cdot g(n_0)$

We will also use the first notation for functions $f, g : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$. In this case, $f = O(g)$ iff there are $n_0, m_0, k \in \mathbb{N}$ such that

$$\forall n, m \in \mathbb{N}\ n \geq n_0 \wedge m \geq m_0 \Rightarrow f(m, n) \leq k \cdot g(m, n)$$

As for the theory of (one-dimensional) automata and formal languages, we use the standard notation and notions that can be found in [4].

Chapter 2

Theory of Picture Languages

2.1 Pictures, Picture Languages

In this section we extend some basic definitions from the one-dimensional theory of formal languages. More details can be found in [20].

Definition 1 *A* picture *over a finite alphabet Σ is a two-dimensional rectangular array (matrix) of elements of Σ, moreover, Λ is a special picture called the* empty picture. *Σ^{**} denotes the set of all pictures over Σ. A* picture language *over Σ is a subset of Σ^{**}.*

Let $P \in \Sigma^{**}$ be a picture. Then, rows(P), resp. cols(P) denotes the number of rows, resp. columns of P (we also call it the *height*, resp. *width* of P). The pair rows(P) $\times$ cols(P) is called the *size* of P. We say that P is a square picture of size n if rows(P) = cols(P) = n. The empty picture Λ is the only picture of size 0×0. Note that there are no pictures of sizes $0 \times k$ or $k \times 0$ for any $k > 0$. For integers i, j such that $1 \leq i \leq$ rows(P), $1 \leq j \leq$ cols(P), $P(i, j)$ denotes the symbol in P at coordinate (i, j).

Example 1 To give an example of a picture language, let $\Sigma_1 = \{a, b\}$, L be the set consisting exactly of all square pictures $P \in {\Sigma_1}^{**}$, where

$$P(i,j) = \begin{cases} a & \text{if } i+j \text{ is an even number} \\ b & \text{otherwise} \end{cases}$$

Pictures in L of sizes 1, 2, 3 and 4 follow.

$$a \qquad \begin{matrix} a & b \\ b & a \end{matrix} \qquad \begin{matrix} a & b & a \\ b & a & b \\ a & b & a \end{matrix} \qquad \begin{matrix} a & b & a & b \\ b & a & b & a \\ a & b & a & b \\ b & a & b & a \end{matrix}$$

Let P_1 be another picture over Σ of size $m_1 \times n_1$. We say P_1 is a *sub-picture* of P iff there are positive integers c_x,c_y, such that

- $c_x + m_1 - 1 \leq m$ and $c_y + n_1 - 1 \leq n$
- for all $i = 1, \ldots, m_1$; $j = 1, \ldots, n_1$ it holds $P_1(i, j) = P(c_x + i - 1, c_y + j - 1)$

Let $[a_{ij}]_{m,n}$ denote the matrix

$$\begin{matrix} a_{1,1} & \ldots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \ldots & a_{m,n} \end{matrix}$$

We define two binary operations – the *row* and *column concatenation*. Let $A = [a_{ij}]_{k,l}$ and $B = [b_{ij}]_{m,n}$ be non-empty pictures over Σ. The column concatenation $A \oplus B$ is defined iff $k = m$ and the row concatenation $A \ominus B$ iff $l = n$. The products of these operations are given by the following schemes:

$$A \oplus B = \begin{matrix} a_{11} & \ldots & a_{1l} & b_{11} & \ldots & b_{1n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{k1} & \ldots & a_{kl} & b_{m1} & \ldots & b_{mn} \end{matrix} \qquad A \ominus B = \begin{matrix} a_{11} & \ldots & a_{1l} \\ \vdots & \ddots & \vdots \\ a_{k1} & \ldots & a_{kl} \\ b_{11} & \ldots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{m1} & \ldots & b_{mn} \end{matrix}$$

It means $A \oplus B = [c_{ij}]_{k,l+n}$, where

$$c_{ij} = \begin{cases} a_{ij} & \text{if } j \leq l \\ b_{i,j-l} & \text{otherwise} \end{cases}$$

and similarly, $A \ominus B = [d_{ij}]_{k+m,l}$, where

$$d_{ij} = \begin{cases} a_{ij} & \text{if } i \leq k \\ b_{i-k,j} & \text{otherwise} \end{cases}$$

Furthermore, the column and row concatenation of A and Λ is always defined and Λ is the neutral element for both operations.

For languages L_1, L_2 over Σ, the column concatenation of L_1 and L_2 (denoted by $L_1 \oplus L_2$) is defined in the following way

$$L_1 \oplus L_2 = \{P \mid P = P_1 \oplus P_2 \wedge P_1 \in L_1 \wedge P_2 \in L_2\}$$

similarly, the row concatenation (denoted by $L_1 \ominus L_2$):

$$L_1 \ominus L_2 = \{P \mid P = P_1 \ominus P_2 \wedge P_1 \in L_1 \wedge P_2 \in L_2\}$$

The *generalized concatenation* is an unary operation $\bigoplus$ defined on a set of matrixes of elements that are pictures over some alphabet: For $i = 1, \ldots, m$; $j = 1, \ldots, n$, let P_{ij} be pictures over Σ. $\bigoplus [P_{ij}]_{m,n}$ is defined iff

$$\begin{gathered} \forall i \in \{1, \ldots, m\}\ \mathrm{rows}(P_{i1}) = \mathrm{rows}(P_{i2}) = \ldots = \mathrm{rows}(P_{in}) \\ \forall j \in \{1, \ldots, n\}\ \mathrm{cols}(P_{1j}) = \mathrm{cols}(P_{2j}) = \ldots = \mathrm{cols}(P_{mj}) \end{gathered}$$

The result of the operation is $P_1 \ominus P_2 \ominus \ldots \ominus P_m$, where each $P_k = P_{k1} \oplus P_{k2} \oplus \ldots \oplus P_{kn}$. See Figure 2.1 for an illustrative example.

$P_{1,1}$	$P_{1,2}$	$P_{1,3}$
$P_{2,1}$	$P_{2,2}$	$P_{2,3}$

Figure 2.1: Scheme demonstrating the result of $\bigoplus[P_{ij}]_{2,3}$ operation.

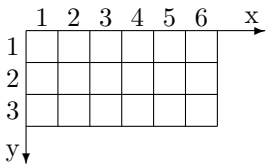


Figure 2.2: The system of coordinates used in our picture descriptions.

For $m, n \in \mathbb{N}$, $\Sigma^{m,n}$ is the subset of Σ^{**} containing exactly all pictures of size $m \times n$.

$$\Sigma^{m,n} = \{P \mid P \in \Sigma^{**} \wedge \mathrm{rows}(P) = m \wedge \mathrm{cols}(P) = n\}$$

A *projection* is every function $\pi : \Sigma \to \Gamma$, where Σ and Γ are alphabets. We extend π on pictures and languages. Let P be a picture of size $m \times n$ over Σ, L be a picture language over Σ. Then

$$\pi(P) = [\pi\,(P(i,j))]_{m,n}$$

$$\pi(L) = \{\pi(P) \mid P \in L\}$$

In our descriptions, we use the system of coordinates in a picture depicted in Figure 2.2. Speaking about the position of a specific field, we use words like up, down, right, left, first row, last row etc. with respect to this scheme.

2.2 Two-dimensional Turing Machines

The two-dimensional Turing machine is the most general two-dimensional automaton. It is a straightforward generalization of the classical one-dimensional Turing machine – the tape consisting of a chain of fields storing symbols from a working alphabet is replaced by a two-dimensional array of fields, infinite in both directions. The additional movements of the head, up and down, are allowed, preserving the possibility to move right and left. We emphasize, in our text, we consider the single-tape model only. However, it is possible to define multi-tape variants as well. A formal definition follows.

Definition 2 *A* two-dimensional Turing machine *is a tuple* $(Q, \Sigma, \Gamma, q_0, \delta, Q_F)$, *where*

- Q *is a finite set of states*

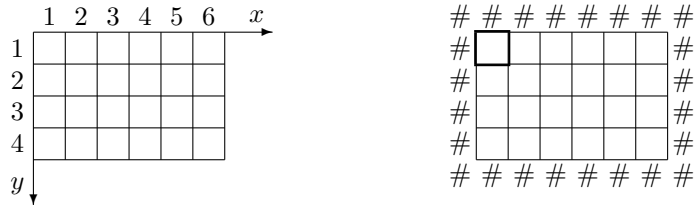


Figure 2.3: Initial configuration – the figure on the left shows coordinates on the tape, where an input picture of size 4×6 is positioned. The figure on the right is the input bordered by #'s, which fill the remaining fields of the tape. The head scans the highlighted field (the top-left corner).

- $\Sigma \subset \Gamma$ *is an input alphabet*
- Γ *is a working alphabet*
- $q_0 \in Q$ *is the initial state*
- $Q_F \subseteq Q$ *is a set of final states*
- $\delta : \Sigma \times Q \to 2^{\Sigma \times Q \times \mathcal{M}}$ *is a transition function* – $\mathcal{M} = \{L, R, U, D, N\}$ *denotes the set of the head movements (left, right, up, down, no movement)*

We assume there is always a distinguished symbol $\# \in \Gamma \backslash \Sigma$ *called the background symbol.*

Comparing to the formal definition of the one-dimensional Turing machine, there is only one difference – the set of the head movements contains two additional elements.

Let $T = (Q, \Sigma, \Gamma, q_0, \delta, Q_F)$ be a two-dimensional Turing machine, $P \in \Sigma^{**}$ be an input picture.

A *configuration* of T is each triple $(q, (x_h, y_h), \tau)$, where

- $q \in Q$ is the current state
- (x_h, y_h) is the coordinate of the head's position
- τ is a mapping $\mathbb{Z} \times \mathbb{Z} \to \Gamma$ assigning a symbol to each field of the tape (the field at coordinate (x, y) stores $\tau(x, y)$). It is required that the subset of all fields storing a symbol different to # is always finite.

The computation of T is defined in the natural way, analogously to computations of Turing machines from the one-dimensional theory. Figure 2.3 shows the initial configuration of T. The head scans the field of coordinate $(1, 1)$. This field stores the top-left corner of the input (assuming $P \neq \Lambda$). During a computation, if T is in a state q and scans a symbol a, the set of all possible transitions is determined by $\delta(a, q)$. If $(a', q', m) \in \delta(a, q)$, then T can rewrite a by a', enter the state q' and move the head in the direction given by m. A computational branch of T halts whenever the control unit reaches some state in Q_F or whenever there is no instruction in δ applicable on the current configuration. The input is accepted if and only if T can reach a state in Q_F.

T is *deterministic* iff $|\delta(a, q)| \leq 1$ for each pair $a \in \Gamma$, $q \in Q$ meaning that at most one instruction can be applied at any given computational step.

T is *#-preserving* iff it does not rewrite any # by another symbol and does not rewrite any symbol different to # by #.

T is *bounded* iff it behaves as follows: Whenever T encounters #, it moves the head back in the next step and does not rewrite the symbol. When the input is Λ, T is bounded iff it does not move the head and rewrite at all.

T is *finite* iff it does not rewrite any symbol.

For an input P, assuming each computational branch of T halts on P, let $t(P)$ be the maximal number of steps T has done among all branches (time complexity for P).

Definition 3 *Let $T = (Q, \Sigma, \Gamma, q_0, \delta, Q_F)$ be a two-dimensional Turing machine and $t(P)$ be defined for all $P \in \Sigma^{**}$. Then, T is of* time complexity

$$t_1(m, n) = \max_{P \in \Sigma^{m,n}} t(P)$$

In the literature, time complexity of a non-deterministic Turing machine is sometimes defined even in cases when the machine accepts, but there are also non-halting branches. However, for our purposes, the definition we have presented is sufficient, we will not work with such computations.

Let T be of time complexity $t_1 : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$. When we are not interested in the dependency of time on picture size, we also define time complexity $t_2 : \mathbb{N} \to \mathbb{N}$ simply as follows

$$t_2(k) = \max_{m \cdot n = k} t_1(m, n)$$

In this case, time complexity depends on the number of input fields only.

2.3 Notes on Computational Techniques

In this section we present two computational techniques and terms related to them. They will allow us to simplify descriptions of algorithms for Turing machines that we will design. The first term is the *marker*, the second term is the *block*.

Working with markers is a commonly used technique. By a marker we mean a special symbol marking a field during some parts of a computation. To be more precise, let $T = (Q, \Sigma, \Gamma, q_0, \delta, Q_F)$ be a Turing machine. T can be extended to have possibility to mark any field of the tape by the marker M in the following way: The working alphabet Γ is replaced by $\Sigma \cup (\Gamma \times \{0, 1\})$ (we assume $\Sigma \cap (\Gamma \times \{0, 1\}) = \emptyset$). Now, if a field stores (a, b), where $a \in \Gamma$ and $b \in \{0, 1\}$, then b of value 1 indicates the presence of M in the field. A field storing a symbol from the input alphabet Σ is considered not to be marked. It is possible to extend T to support any constant number of different, mutually independent markers, let us say $M_1, \ldots, M_k$. The working alphabet needs to be extended to $\Sigma \cup \Gamma \times \{0, 1\}^k$ in this case.

It is quite often suitable to organize the content of a working tape into blocks that are rectangular areas of tape fields. Formally, let $T = (Q, \Sigma, \Gamma, q_0, \delta, Q_F)$

l,t	t	t	t	t	t	t,r
l						r
l,b	b	b	b	b	b	b,r

Figure 2.4: A block represented using markers t, l, b and r that are placed in the top, left, bottom, resp. right part of the perimeter.

be a two-dimensional Turing machine and let $f_{x,y}$ denote the tape field at coordinate (x, y). Then, for any tuple of integers (c_1, c_2, s_1, s_2), where $s_1, s_2 > 0$, the set

$$B = \{f_{x,y} \,|\, c_1 \leq x < c_1 + s_1 \wedge c_2 \leq y < c_2 + s_2\}$$

is a block. We say, B is *given by* (c_1, c_2, s_1, s_2).

Note that (c_1, c_2) is the coordinate of the top-left corner of B, s_1, resp. s_2 the number of columns, resp. rows. We can treat blocks like pictures and write $\text{rows}(B) = s_2$, $\text{cols}(B) = s_1$. We can also assign a content to a block, but, comparing to pictures, this assignment is a function of T's configuration. For $i = 1, \ldots s_1$; $j = 1, \ldots, s_2$, let $B(i, j)$ be the field in B of coordinate $(c_1 + i - 1, c_2 + j - 1)$ (i.e. (i, j) is a relative coordinate within the block). Furthermore, let $s_T(f, c)$ be the symbol of Γ stored in the field f in the configuration c. Then, the picture assigned to B in c is P, where $P(i, j) = s_T(B_{i,j}, c)$.

The *perimeter* (or *border*), denoted here by B', of B is the subset of its fields given as follows:

$$B' = \{B(i, j) \,|\, i = 1 \vee j = 1 \vee i = \text{cols}(B) \vee j = \text{rows}(B)\}$$

When working with blocks during a computation, it is usually useful and sufficient to represent a block by markers placed in the border fields (i.e. fields of B') as it is shown in Figure 2.4.

Let B_1, B_2 be two blocks, B_1 given by (x_1, y_1, s, t), B_2 by $(x_1 + d, y_1, s, t)$, where $d \geq t$. It means B_2 is located right to B_1 and both blocks have the same number of rows and columns. Let us assume the borders of both blocks are marked. By these circumstances, let us solve the task how to copy the content of B_1 into B_2.

An appropriate Turing machine T can work as follows: It starts having the head placed on the top-left field of B_1 (we assume T is able to locate this field). The block is copied row by row, each row field by field. Two markers are used to mark the current source and destination. T records the scanned symbol a in states, marks the scanned field f as 'read', moves rightwards until it encounters the top-left field of B_2 (it can be found thanks to border markers), writes a to the detected field, marks the field as 'written', moves back (leftwards) to f, clears the 'read' marker, moves to the right neighbor of f (which is the second field in the scanned row of B_1), marks it 'read', stores the correspondent symbol in states again, moves rightwards until the field marked 'written' is detected, clears the marker, moves right by one field, copies the symbol recorded in states, places the marker 'written', returns back to the field marked 'read' and so on. When T reaches the end of the first row of B_1 (at the moment the last field of

the row has been copied), it goes to the last field of the second row and starts to copy fields of this row in the reversed order (from right to left). In addition, it does not mark by 'written' the field copied as the last one in the currently processed row of B_2, since this marker is not needed for the next iteration. The procedure is repeated until all rows have been copied. The order in which fields are being copied is changed each time T finishes a row.

As for the time complexity of the algorithm, one iteration (to copy a field) requires $2d + 3$ computational steps – two steps to check if the current source field is the last one to be copied in the current row, $d - 1$ movements right to detect the field marked 'written', one movement from this field to go to the proper destination field, d movements back to the source field and one movent to the source field of the next iteration. When the algorithm ends, the head is placed on the bottom-right corner of B_1 (T need not to move the head to the next source field, when it returns back from B_2 to B_1 and the bottom-right field is the source field). Summing steps over all iterations, we see that the number of steps of the algorithm is not greater than $s \cdot t \cdot (2d + 3)$.

B_1 can be copied into B_2 also if B_2 is given by (x_1+d, y_1, s, t). The procedure is analogous, time complexity remains the same. Even a general case, when B_2 is given by (x_2, y_2, s, t), can be handled similarly. For T, it is only necessary to be able to locate the i-th row of B_2 when scanning the i-th row of B_1 and vice versa. It can be achieved by placing suitable markers before the procedure is launched. Moreover, if B_1 and B_2 overlap, the working alphabet is required to code content of two block's fields. Nevertheless, assuming the number of fields of the blocks is fixed, time complexity of the procedure is linear in the distance (measured in fields) between the top-left corners of B_1 and B_2 - the distance is $|x_1 - x_2| + |y_1 - y_2|$.

We will refer to the described technique using the term *copying a block field by field*.

Chapter 3

Two-dimensional Finite-state Automata

3.1 Finite-state Automaton

Definition 4 *A* two-dimensional finite-state automaton *is every tuple* $(Q, \Sigma, q_0, \delta, Q_F)$*, where* $(Q, \Sigma, \Sigma \cup \{\#\}, q_0, \delta, Q_F)$ *is a two-dimensional finite bounded Turing machine.*

As we have already mentioned in section 1.1, we abbreviate a two-dimensional finite-state automaton by FSA, a deterministic FSA by $DFSA$ (we follow the notation in [20]). We use these abbreviations prior to one-dimensional finite-state automata. If we need to refer them, we will emphasize it explicitly.

Since a FSA does not perform any rewriting, its working alphabet consists of elements in the input alphabet and the background symbol, thus it is fully given by a tuple of the form $(Q, \Sigma, q_0, \delta, Q_F)$.

Finite-state automata have been studied by many authors. FSA's working over pictures consisting of one row behave like two-way one-dimensional finite-state automata that are of the same power as one-way one-dimensional finite-state automata. It means, FSA's can be considered as a generalization of one-dimensional finite-state automata. The question that arises is what are the properties of the class $L(FSA)$. Is this class a good candidate for the base class of the theory of two-dimensional languages, analogously to the class of regular languages? We list some of the most important properties of $L(FSA)$ and FSA's (as they are given in [20]).

- $L(FSA)$ is not closed under concatenation (neither row or column)
- $L(FSA)$ in not closed under complement
- $L(DFSA) \neq L(FSA)$
- The emptiness problem is not decidable for FSA's.

Comparing to the class of regular languages, all these properties are different.

Example 2 To demonstrate capabilities of FSA's, let us define the following language over $\Sigma = \{a, b\}$.

$$L = \{A \oplus B \mid A \in \{a\}^{**} \wedge B \in \{b\}^{**} \wedge \text{rows}(A) = \text{cols}(A) = \text{rows}(B) = \text{cols}(B)\}$$

L contains exactly each picture of size $n \times 2n$ ($n \in \mathbb{N}$) consisting of two parts - the left part is a square of a's, while the right part a square of b's. We show that L is in $L(DFSA)$. Let P be an input, A be a $DFSA$ we construct. It can easily perform these verifications:

- Checks if all rows are of the form $a^i b^j$ for some global constants i, j. To do it, A scans row by row, in a row, it verifies if there is a sequence of a's followed by a sequence of b's and, starting by the second row, whenever it detects the end of the sequence of a's, it checks if the correspondent sequence of a's in the row above ends at the same position - this can be tested locally.
- Moves the head to the top-left corner, then, moves it diagonally (one field right and one field down repeatedly) until the last row is reached. In the field reached in the last row, A checks whether it contains a.
- Moves the head right by one field, checks if the field contains b, moves diagonally right and up until the first row is reached. Finally, A checks if the movement has ended in the top-right corner of P.

3.2 Finite-state Automata over One-symbol Alphabet

In this section we will study FSA's working over one-symbol alphabets. We prove a theorem allowing to show that some specific picture languages over one-symbol alphabet cannot be recognized by a FSA. As a consequence, we prove that $L(FSA)$ is not closed under projection. As the properties we have listed in the previous section, this result is also different comparing to the class of regular languages.

For a one-symbol alphabet Σ (where $|\Sigma| = 1$), we use $[m, n]_\Sigma$ to denote the only picture over Σ of size $m \times n$. When it is clear from the context which alphabet Σ is referred, we simply write $[m, n]$.

We can prove a kind of pumping lemma for picture languages from $L(FSA)$ over one-symbol alphabets.

Proposition 1 *Let A be a FSA working over a one-symbol alphabet Σ. Let A have k states. If A accepts a picture P such that $m = \text{rows}(P)$, $n = \text{cols}(P) \geq 1 + k \cdot m$, then, for all $i \in \mathbb{N}$, A accepts $[m, n + i \cdot (m \cdot k)!]$ too.*

Proof. Let $\mathcal{C} = \{C_i\}_{i=0}^{t}$ be an accepting computational branch of A. A configuration of A is fully determined by the head position, the current state and size $m \times n$. Since m, n are fixed during the whole computation, we consider elements of $\mathcal{C}$ to be triples (i, j, q), where $i \in \{0, \ldots, m+1\}$, resp. $j \in \{0, \ldots, n+1\}$ is a horizontal, resp. vertical position of the head and $q \in Q$. Note that A can move the head one symbol out of the area of P, thus i, resp. j can be 0 or $m + 1$, resp. $n + 1$.

Let $C' = (r', n, q')$ be the configuration in $\mathcal{C}$ in which the head of A reaches the last column of P first time. (If the head never reaches the last column it is evident that A accepts every picture $[m, n + l]$ for any integer l, so the proof can be easily finished here in this case.) Next, let $\mathcal{C}_1$ be the only contiguous subsequence of $\mathcal{C}$ satisfying

- the first element is a configuration, in which the head scans a field of the first column of P
- no other element in $\mathcal{C}_1$ (except the first one) is a configuration, where the head scans a field of the first column
- the last element of $\mathcal{C}_1$ is C'

Finally, we define $\mathcal{C}_2$ to be a subsequence of $\mathcal{C}_1$ (not necessary contiguous) of length $m \cdot k+1$, where each i-th element ($i = 1, \ldots, m \cdot k+1$) is the configuration among elements of $\mathcal{C}_1$, in which the head of A reaches the i-th column first time. Since $|\mathcal{C}_2| = k \cdot m+1$, there are at least two configurations in $\mathcal{C}_2$ such that the head is placed in the same row and A is in the same state. Let these configurations be $C_{t_1} = (r, c_1, q)$ and $C_{t_2} = (r, c_2, q)$, where $t_1 < t_2$ and thus $c_1 < c_2$. We emphasize $\mathcal{C}_2 \subseteq \mathcal{C}_1$ guaranteers that, during the part of the computation starting by the t_1-th step and ending by the t_2-th step, the head of A can never scan the symbol # located in the column left to P.

Let $C_{t_3} = C'$ and $p = c_2 - c_1$. Now, let us consider what happens if P is extended to $P' = [m, n+p]$ and A computes on P'. There exists a computational branch that reaches the last column of P' first time after $t_3 + t_2 - t_1$ steps entering the configuration $(r', n + p, q')$. A can compute as follows. It performs exactly the same steps that are determined by the sequence $C_0, C_1, \ldots, C_{t_2}$. After that it repeats steps done during the $C_{t_1}, \ldots, C_{t_2}$ part, so it ends after $t_2 - t_1$ additional steps in the configuration $(r, c_2 + p, q)$. Next, it continues with $C_{t_2}, \ldots, C_{t_3}$, reaching finally the desired configuration.

Instead of P', it is possible to consider any extension of P of the form $[m, n + b \cdot p]$ for any positive integer b. A can reach the last column in the state q' having its head placed in the r'-th row again. It requires to repeat the $C_{t_1}, \ldots, C_{t_2}$ part b times exactly.

Furthermore, we can apply the given observation on next computational steps of A repeatedly. For A computing over P, we can find an analogy to $\mathcal{C}_2$ for steps following after the t_3-th step. This time A starts scanning the last column of P and the destination is the first column. We can conclude again that, for some period p', A can reach the first column in the same row and state for all pictures of the form $[m, n + b \cdot p']$, $b \in \mathbb{N}$. Then, there follows a part, when A starts in the first column and ends in the last column again, etc. Since a period is always less than $m \cdot k + 1$, $(m \cdot k)!$ is surely divisible by all periods. Hence, if a picture $[m, n + i \cdot (m \cdot k)!]$ is chosen as an extension of P, then, there is a computation of A fulfilling: whenever A reaches the first or the last column, it is always in the same state and scans the same row in both cases (meaning P and the extended picture). Finally, during some movement of the head between the border columns of P, if A accepts before it reaches the other end, then it accepts the extended picture as well. □

Note that it is possible to swap words "row" and "column" in the lemma and make the proof analogously for pictures P fulfilling $\text{rows}(P) \geq 1 + k \cdot \text{cols}(P)$.

We have found later that Proposition 1 has been already proved ([11]), thus this result is not original.

A stronger variant of Proposition 1 can be proved for two-dimensional deterministic finite-state automata.

Proposition 2 *Let A be a $DFSA$ working over a one-symbol alphabet Σ. Let A have k states. If A accepts a picture P such that $m = \mathrm{rows}(P)$, $n = \mathrm{cols}(P) \geq 1 + k \cdot m$, then, there is an integer s, $0 < s \leq k! \cdot (mk)^{2k}$ such that, for any $i \in \mathbb{N}$, $[m, n + i \cdot s]$ is accepted by A too.*

Proof. The main idea of the proof remains the same as it was presented in the proof of Proposition 1. The only different part is the estimate of the number of all possible periods. We show that in the case of $DFSA$ we can encounter at most $3k$ different periods. Moreover, k of these possible periods are less then or equal to k (the remaining periods are bounded by $m \cdot k$ again).

Let us consider a part of the computation of A that starts and ends in the same row and state, that never reaches left or right border column of #'s and that is the minimal possible with respect to the period, i.e. it cannot be shortened to a consecutive subsequence to obtain a shorter period. Let the steps of the considered part of the computation be given by a sequence of configurations $\mathcal{C} = \{C_i\}_{i=t_1}^{t_2}$, where $C_i = (r_i, c_i, q_i)$. We have $r_{t_1} = r_{t_2}$, $q_{t_1} = q_{t_2}$.

We investigate two situations:

1) The head of A does not reach the first or last row of P during $\mathcal{C}$

2) The head of A reaches the first or last row of P

Ad 1). We show that the period cannot be longer than k. By contradiction, let $t_2 - t_1 > k$. Then, there is a pair $C_i, C_j \in \mathcal{C}$, $i < j < t_2$, such that the states of A related to these configurations are not the same ($q_i \neq q_j$). Now, if $r_i = r_j$, it contradicts to the minimality of the period of $\mathcal{C}$. Otherwise, if $r_i \neq r_j$, let us say $r_i < r_j$ (the second case is similar), then after performing $j - i$ steps, the vertical coordinate of the head's position is increased by $r_j - r_i$ and A is again in the same state. It means, the vertical coordinate is being incremented in cycles of length $j - i$ till the bottom of P is reached, which is a contradiction.

Ad 2). Since A is deterministic, the length of the period, i.e. $t_2 - t_1$, is uniquely determined by the state A is in when it reaches the top, resp. bottom row of P. It means there are at most $2k$ such periods.

All periods of the first type divide $k!$. Each period of the second type is bounded by $m \cdot k$, thus all these periods divide a positive integer b less than $(mk)^{2k}$. It means, for each $i \in \mathbb{N}$, $[m, n + i \cdot k! \cdot b]$ is accepted by A.

□

Definition 5 *Let Σ be a one-symbol alphabet, $f : \mathbb{N}^+ \to \mathbb{N}^+$ be a function. We say that a language L over Σ* represents *f if and only if*

1. $\forall m, n \in \mathbb{N}^+ \, [m, n]_\Sigma \in L \Leftrightarrow n = f(m)$

2. $\Lambda \notin L$

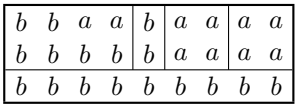


Figure 3.1: Picture P_3.

For a fixed Σ such that $|\Sigma| = 1$, we denote the language representing f by L_f, i.e.

$$L_f = \{[n, f(n)]_\Sigma \mid n \in \mathbb{N}^+\}$$

We also say *f is represented by L_f*.

Theorem 1 *Let $f : \mathbb{N}^+ \to \mathbb{N}^+$ be a function such that $f = o(\text{lin})$. Then, L_f is not in $L(FSA)$.*

Proof. By contradiction. Let A be a FSA recognizing L_f and let k be the number of states of A. Since $f = o(\text{lin})$, there is an integer n_0 such that $f(n_0) \geq (k+1) \cdot n_0 \geq k \cdot n_0 + 1$. By Proposition 1, it holds $[n_0, f(n_0) + (k \cdot n_0)!] \in L_f$. It is a contradiction. □

Example 3 Let L be a language consisting exactly of all pictures over $\Sigma = \{a\}$ having the number of columns equal to the square of the number of rows. Formally, $L = \{[n, n^2] \mid n \in \mathbb{N}^+\}$. Theorem 1 implies $L \notin L(FSA)$.

We shall note again that the result given in Example 3 can be found in the literature ([6]).

Example 4 We define a recurrent sequence of pictures $\{P_i\}_{i=1}^{\infty}$ over the alphabet $\Sigma = \{a, b\}$.

1. $P_1 = b$
2. For all $n \geq 1$,
$$P_{n+1} = (P_n \oplus V_n \oplus S_n) \ominus H_{n+1}$$
where S_n the rectangle over $\{a\}$ of size $n \times 2n$, V_n is the column of b's of length n and H_{n+1} is the row of b's of length $(n+1)^2$.

Note that for every picture P_n it holds $\text{rows}(P_n) = n$, $\text{cols}(P_n) = n^2$ (this result can be easily obtained by induction on n). We define $L = \{P_n \mid n \in \mathbb{N}^+\}$. Figure 3.1 shows an example of a picture in L.

Lemma 1 *The language L in Example 4 can be recognized by a $DFSA$.*

Proof. We describe how to construct a $DFSA$ A recognizing L. The computation is based on gradual reductions of an input P to sub-pictures. If $P = P_{n+1}$ for some positive integer n, then the sequence of sub-pictures produced by the process is $P_n, P_{n-1}, \ldots, P_1$. Moreover, the produced sequence is always finite and P_1 is its last element if and only if $P \in L$.

Let us take a closer look at the procedure performing one reduction. We consider the content of the tape to be as it is shown in Figure 3.2. The input

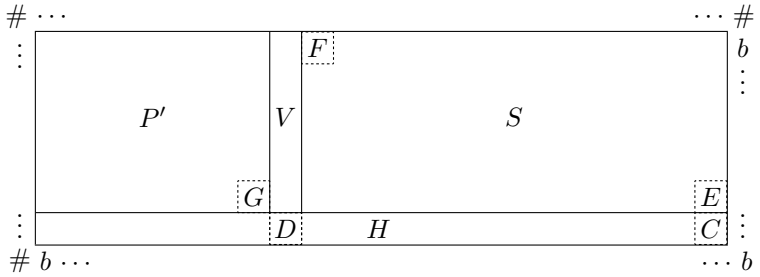


Figure 3.2: Division of P into sub-pictures. If P should be in L, S is required to be a rectangle of size $k \times 2k$, V a column of b's and H a row of b's. P' is the product of the procedure. C,D,E,F and G are important fields of P referred in our description.

to the procedure is the picture P. The head scans the bottom-right corner of P (denoted by C). P is bounded by the background symbol # on the left and top side, the right and bottom borders are formed of b's. The goal of the procedure is to divide P into sub-pictures P', S, V and H.

A computes as follows (note that A halts immediately and rejects the input if it founds that one of the listed conditions is not satisfied). First of all, A moves the head left until the end of H is reached. During this movement, it verifies if all symbols of H are b's. Moreover, it checks if the row above H is of the form $b^i a^j$, $i, j \geq 1$. (It means A does not perform pure movement to the left – it goes up and down, before it moves left by one field.) The last row of P' must contain b's only, otherwise P' cannot be equal to some P_i. When A finishes the verification, it is able to find the field D, since it is the leftmost field of H having the top-right neighboring field that contains a. A continues by moving the head to D and then goes trough all fields of V to ensure all of them contain b. After that, A moves its head to the top-right neighbor of D and starts to verify if all fields of S contain a. It is done column by column. A moves up until # is scanned. Then, the head is moved vertically back to H followed by one movement to the right. A continues by checking the second column, etc. Now, to complete the verification of S requires to check whether it is a rectangle of size $k \times 2k$ ($k \in \mathbb{N}^+$). A places the head over E, then, moves it diagonally to F (it performs three movements repeatedly – left, left and up). Reaching the border exactly in F indicates that S is of the required size.

Finally, A moves the head to G. At this moment, the whole procedure is finished and A is ready to perform it again (on P').

Before the process of reductions can be launched, A must move the head to the bottom-right corner of the input. The process ends if A detects that some picture P cannot be divided correctly into sub-pictures as it was described in the previous paragraphs, or if P_1 is the result of some reduction. A can detect P_1 when the head is placed over the field G. If G is the top-left corner of the input and it contains b, then P_1 has been obtained. One more remark – note that when A performs the first reduction and the whole input to A is taken as an input to the procedure, then it is bounded by #'s only, there are no b's. However, it should be obvious that this case can be distinguished and handled

easily. □

It is a known fact that the class of one-dimensional regular languages is closed under homomorphism and since a projection is a special case of homomorphism, the class is closed under projection as well. We can show that the classes $L(FSA)$, $L(DFSA)$ do not share this property.

Theorem 2 *The classes $L(FSA)$, $L(DFSA)$ are not closed under projection.*

Proof. Let L_1 be the language in Example 3 and L_2 the language in Example 4. Let us define a projection $f : \{a, b\} \to \{a\}$ such that $f(a) = f(b) = a$. We shall see that $f(L_2) = L_1$. Since L_2 can be recognized by a deterministic finite-state automaton and L_1 cannot be recognized by a non-deterministic finite-state automaton, the theorem is proved. □

3.3 Simulation of Finite-state Automata by Deterministic Turing Machines

In this section we study how for a given two-dimensional non-deterministic finite-state automaton to construct a two-dimensional deterministic bounded Turing machine recognizing the same language. Our goal is to obtain machines optimized with respect to time complexity.

Questions regarding a relation of FSA's to determinism were also studied by P. Jiricka and J. Kral in [10], where the simulation of FSA's by two-dimensional deterministic forgetting automata is presented. However, in this case, the aim was to show the existence of such a simulation rather than to deal with time complexity.

We start with a relatively simple and straightforward simulation of FSA's.

Proposition 3 *Let A be a FSA recognizing L in time $t(m, n)$. There is a bounded deterministic TM T such that T recognizes L in time $O(t(m, n) \cdot m \cdot n)$.*

Proof. We will construct a DTM T. Let $A = (Q, \Sigma, q_0, \delta, Q_A)$ and let P be an input picture of size $m \times n$. We can assume P is non-empty, since T can decide immediately if the empty picture should be accepted or not.

During the computation, T records a subset of Q in each field of the portion of the tape containing P. For a field, the subset represents states in which A can reach the field. At the beginning, all fields store empty subsets except the top-left corner which stores $\{q_0\}$. Subsets are updated in cycles. During one cycle, T goes through all fields of P, e.g. row by row. When scanning a field f, it reads the subset currently recorded in f (let it be Q_f) and considers all transitions that A can perform in one step when it is in a state in Q_f having its head placed over f. By these all possible steps, T updates Q_f and the subsets recorded in neighbors of f to which the head can be moved (in another words, T adds new states detected to be reachable).

Whenever T encounters that a state in Q_A is reachable in some field, P is accepted. During a cycle, T memorizes in its state, whether it has updated at least one subset. If no update has been done, it indicates that all reachable

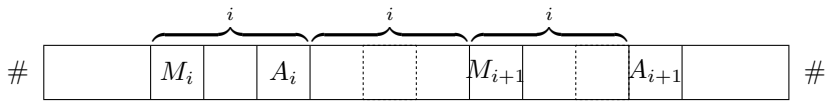


Figure 3.3: A portion of the tape containing markers A_i, M_i, A_{i+1}, M_{i+1}. The distance between A_i and A_{i+1} is $2 \cdot i + 1$.

pairs of state and position have been already detected, thus T halts and rejects P.

It requires time $O(m \cdot n)$ to complete one cycle. We can see that if A can reach some field f in state q in time t_1, then T detects the pair (f, q) to be reachable by the t_1-th cycle has been finished. It implies $t(m, n) + 1$ cycles are required to calculate subsets of reachable states maximally, thus P is correctly accepted or rejected in time $O(t(m, n) \cdot m \cdot n)$. □

Lemma 2 *It is possible to construct a one-dimensional deterministic bounded TM T that for any non-empty string w over an alphabet Σ computes the number $k = \lfloor \sqrt{|w|} \rfloor$ and represents it by a marker placed in the k-th field of w. Moreover, T works in time $t(n) = O(n^{\frac{3}{2}})$.*

Proof. Let w be an input of length $n > 0$. We construct T of the required type computing $\lfloor n \rfloor$.

Let $\mathcal{S} = \{a_i\}_{i=1}^{\infty}$ be a sequence, where $a_i = i^2$ for each $i \in \mathbb{N}^+$. The idea of the computation is to successively place markers $A_1, A_2, \ldots, A_k$ on fields of w, each A_i to be in the distance a_i from # precessing the leftmost field of w. It is evident that if $k = \lfloor \sqrt{n} \rfloor$, then A_k is the rightmost marker still positioned in w.

Except markers A_i, T will also use auxiliary markers M_i. For each A_i, the marker M_i is placed in the distance $a_i - 1$ to the left from A_i. Note that all markers A_i are represented using the same symbol, A_i denotes just an occurrence of the symbol on the tape (similarly for M_i's).

The computation starts by placing A_1 and M_1 on the first field of w. Positions of next markers are determined inductively. Let us assume A_i and M_i are correctly placed on the tape. M_{i+1} should be in the distance

$$(i + 1)^2 = i^2 + 2 \cdot i + 1 = a_i + 2 \cdot i + 1$$

It means, it is sufficient to copy the block starting by M_i and ending by A_i after A_i two times. A_{i+1} is putted after the second copy of the block, M_{i+1} is placed on the first field of the second copy. All is illustrated in Figure 3.3.

If T encounters the end of w, it immediately stops placing markers and starts to count how many markers A_i there are on the tape. It means, T moves the head to the first field of w, removes A_1 and initializes by 1 an unary counter, which will represent the desired value in the end. Then, T repeatedly moves right until it detects some marker A_i, removes it, moves back and increments the unary counter. When T encounters the end of w, all A_i's have been counted, so T halts.

It remains to estimate time complexity of the computation. Assuming T has placed A_i, M_i and the head is scanning the field containing A_i, to place the next

pair of markers requires time $c_1 \cdot i^2 + c_2$, where c_1 and c_2 are suitable constants. The first copy of the block is created copying field by field in time $c_1 \cdot i^2$. The second copy is created using the first copy, time is the same. M_{i+1} is marked after the the first copy is created, A_{i+1} after the second copy is created. Both require constant time c_2. We derive

$$\sum_{i=1}^{k+1} c_1 \cdot i^2 + c_2 \leq c_1 \cdot \sum_{i=1}^{k+1} (k+1)^2 + c_2 = c_1 \cdot (k+1)^3 + c_2 \cdot (k+1) = O\left(n^{\frac{3}{2}}\right)$$

When T counts the number of markers A_i during the second phase, one marker is processed in time at most n, thus the total time of this phase is $O(n \cdot k) = O(n^{\frac{3}{2}})$ again. □

Lemma 3 *It is possible to construct a two-dimensional deterministic bounded TM T computing as follows. Let P be an input picture to T, next, let*

$$m = \min(\text{cols}(P), \text{rows}(P))$$
$$n = \max(\text{cols}(P), \text{rows}(P))$$
$$k = \lfloor \sqrt{n} \rfloor$$

T checks whether $k \leq m$. If so, it represents k in unary in the first row, resp. column (depending on which of these two is of a greater length). Otherwise it halts. Moreover, T is of time complexity $O\left(\min(m,k)^3\right)$.

Proof. First of all, T compares values $\text{cols}(P)$ and $\text{rows}(P)$. It starts scanning the top-left corner. Then it moves the head diagonally, performing repeatedly one movement right followed by one movement down. Depending on during which of these two movements T reaches the background symbol first time, it makes the comparison.

Without loss of generality, let $m = \text{rows}(P) \leq \text{cols}(P) = n$. T marks the m-th field of the first row. Again, this field is detected moving the head diagonally, starting in the bottom-left field. After that T uses the procedure of Lemma 2 to compute k, but this time with a slight modification – whenever a field is marked by some A_i, T increases the counter immediately (it does not wait till the end to count all A_i's – it should be clear that this modification does not have an impact on time complexity). If the counter exceeds m, T stops the computation.

To compare the number of rows and columns as well as to mark the distance m requires time $O(m)$. If $k \leq m$, then T computes k in time $O(n^{\frac{3}{2}}) = O(k^3)$, otherwise it takes time $O(m^3)$ to exceed the counter. It implies T is of time complexity $O\left(\min(m,k)^3\right)$. □

Proposition 4 *Let $A = (Q, \Sigma, q_0, \delta, Q_A)$ be a FSA recognizing L over Σ. It is possible to construct a two-dimensional deterministic bounded Turing machine T that for each picture P over Σ of size $m \times n$, where $\min(m,n) \geq \lfloor \max(m,n)^{\frac{1}{2}} \rfloor$, decides whether P is in L in time $t(m,n) = O(\min(m,n) \cdot \max(m,n)^{\frac{3}{2}})$.*

Proof. Let $A = (Q, \Sigma, q_0, \delta, Q_A)$ be a FSA recognizing L, P be an input picture over Σ. We denote $m = \text{rows}(P)$ and $n = \text{cols}(P)$. T detects in time $O(m+n)$,

which of these numbers is greater (if any). It is again done by moving the head diagonally in the right-down direction, starting at the top-left field. Without loss of generality, let $m \leq n$. By the assumption of the theorem, we have $m \geq \lfloor n^{\frac{1}{2}} \rfloor$.

Comparing to the construction presented in the proof of Proposition 3, we will use a different strategy now. We will work with blocks (parts of P) storing a mapping. For a block B, the related mapping will provide the following information (B' denotes the perimeter of B).

- For each pair (f, q), where $f \in B'$ and $q \in Q$, the mapping says at which fields and in which states A can leave B if it enters it at the field f, in the state q (to be more precise, when we speak about leaving the block, we mean, when A moves the head outside B first time, after it has been moving it within the block).
- For each pair (f, q), the mapping also says if A can reach an accepting state without leaving B, by the assumption it has entered it at the field f, in the state q.

If the whole area of P is divided into a group of disjunct blocks, it is sufficient to examine movements of the head across the blocks only to find out whether A accepts P. We will need to solve what size of the blocks is optimal and how to compute the mappings.

First of all, we will give details on how a mapping is stored in B. Let

$$\max(\text{rows}(B), \text{cols}(B)) \leq 2 \cdot \min(\text{rows}(B), \text{cols}(B)) - 1 \quad (1)$$

For $i = 1, \ldots, \text{rows}(B)$, resp. $j = 1, \ldots, \text{cols}(B)$, let R_i, resp. C_j be the i-th row, resp. j-th column of B. Three functions, we are going to define now, form an equivalent to the mapping. They will also help to make the text more readable.

- $S : B' \times Q \to 2^{B' \times Q}$, where $S(f, q)$ is the set containing exactly each $(f', q') \in B' \times Q$ such that A in q scanning f can reach f' in the state q' without leaving the area of B.
- $\text{acc} : B' \times Q \to \{\text{true}, \text{false}\}$, where $\text{acc}(f, q) = \text{true}$ iff A in q scanning f can reach some state in Q_A without leaving B.
- $s : B' \times Q \times B' \to 2^Q$, where

$$s(f, q, h) = \{q' \mid (h, q') \in S(f, q)\}$$

A mapping is fully determined by S and acc, since, if T knows states reachable in a field of B', it can easily compute in which states A can leave B from this field. For each pair $f \in B'$, $q \in Q$, it could possibly hold that $|S(f, q)| = |B'| \cdot |Q|$, thus it is clear that $S(f, q)$ cannot be stored in f directly. A space linear in $|B'|$ must be assigned to it. Figure 3.4 shows one possible solution of how to achieve it.

Let us assume f is the s-th field of the first row of B as it shown in Figure 3.4. Moreover, let $\text{cols}(B) \leq 2 \cdot \text{rows}(B) - 1$. Then, C_s and R_r, where

$$r = \begin{cases} s & \text{if } s \leq \text{rows}(B) \\ 2 \cdot \text{rows}(B) - s & \text{otherwise} \end{cases}$$

store values of S for f and each $q \in Q$ as follows:

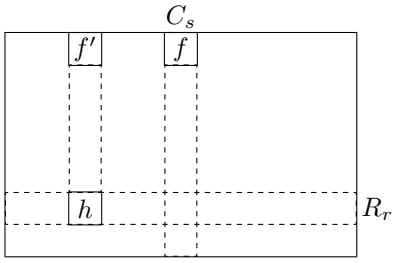


Figure 3.4: Assuming f is the s-th field in the first row, functional values related to this field are stored in the row R_r and the column C_s. E.g., for f' and each $q \in Q$, the field h stores $s(f, q, f')$.

- C_s, resp. R_r is used to represent vectors of length $|C_s|$, resp. $|R_r|$. Each component of these vectors is of a value which is a subset of Q. For example, let us consider R_r. For fixed f and each $q \in Q$, $|Q|$ pairs of vectors are needed – within a pair, the first vector corresponds to the top part of B', while the second one to the bottom part of B'. Let f' be the i-th field in the first row, $\mathcal{V}$ be the first vector of the pair related to f, and q. Then, the i-th component of $\mathcal{V}$ is of value $s(f, q, f')$ and it is represented in the i-th field of R_r (see Figure 3.4).
- For each $q \in Q$, $\mathrm{acc}(f, q)$ can be stored directly in f.

The remaining three variants (fields in the right, resp. left, resp. bottom part of B') use analogous representations. Note that if T scans f, it can easily search through fields of C_s and R_r. T reaches R_r if it moves the head diagonally in the left-down direction until it encounters the left border of B – then R_r is found as well – or the bottom border – in this case T moves left-up until the left border is reached which occurs in the row R_r (T enters definitely the right border during the second movement, since $\mathrm{cols}(B) \leq 2 \cdot \mathrm{rows}(B) - 1$). The detection of C_s and R_r is done in time linear in $|C_s| + |R_r|$.

We need to verify that the presented usage of space in B requires each field of B to store a constant number of bits only. Let h be a field in B belonging to the i-th row and j-th column. Let us discuss how many fields of R_1 use h as a part of their storage. These fields are at most three

- the j-th field of R_1 uses h to store components of the vertical vectors
- the i-th field of R_1 uses h to store components of the horizontal vectors
- if $1 \leq 2 \cdot \mathrm{rows}(B) - i \leq \mathrm{cols}(B)$, then the field of R_1 of index $2 \cdot \mathrm{rows}(B) - i$ uses h also to store components of the horizontal vectors

Since the border is formed of four parts, it implies h cannot be shared by more than 12 fields of B' as a part of their storage.

Now, we show that it is possible to merge mappings stored in four neighboring blocks to obtain the mapping for the whole area. We consider the blocks to be as they are depicted in Figure 3.5. We assume that the size of each block as well as of $B = \bigoplus [B_{ij}]_{2,2}$ fulfills condition (1). The mapping related to B will be computed as a composition of partial mappings – for each pair $f \in B'$,

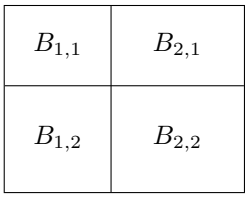


Figure 3.5: Four neighboring blocks storing mappings to be merged.

$q \in Q$, we are looking for values $S(f, q)$, $\mathrm{acc}(f, q)$. We are ready to describe the computation performing this task. In the description, we index S, acc, s by a block the functions are related to. For example, we write S_B to denote the function for the block B.

Let f_1 be a field belonging to the perimeter of one of the blocks. During the procedure, two sets are assigned to each of these fields. For f_1, we denote them by $Q_P(f_1)$ and $Q_N(f_1)$. Contents of these two sets are represented in f_1 and they are being updated consecutively during the procedure. After each update, it always holds $Q_P(f_1) \cap Q_N(f_1) = \emptyset$ and $Q_P(f_1) \cup Q_N(f_1)$ contains all states detected so far to be reachable in f_1. States in $Q_P(f_1)$ have been processed (we explain later what does it mean), while states in $Q_N(f_1)$ have not been processed yet. T also stores in states a boolean flag F determining whether a state in Q_A has been detected to be reachable (anywhere in B). The value of F corresponds to $\mathrm{acc}_B(f, q)$ at the end.

At the beginning, all sets Q_N, Q_P are empty, except $Q_N(f)$ which is initialized by $\{q\}$. After that, T performs the following steps repeatedly:

1) Assuming the head of T scans the top-left field of B, T tries to find a field g such that $Q_N(g) \neq \emptyset$. To do it, it scans fields belonging to the perimeters of B_{ij}'s until a suitable field g in a block C is found. A state q' in $Q_N(g)$ is chosen. If there is no such a field, T stops to repeat the steps.

2) q' is removed from $Q_N(g)$ and added to $Q_P(g)$ (q' is considered to become processed in g). After that, T searches for fields of the perimeter of C, that are reachable from the configuration g, q'. To do it, it goes through fields storing values related to $S_C(g, q')$. Whenever T encounters, there is a field h in the perimeter of C such that $Q' = s_C(g, q', h) \neq \emptyset$, it performs step 3).

3) T checks transitions from h to neighboring blocks. The head is moved to h and, for each state $q_1 \in Q'$, it is checked whether A in state q_1 scanning h can move its head into other block (one of B_{ij}'s, but not C). If so, let the entered field of the other block be h_2. T updates $Q_N(h_2)$ by adding all states in which A being in the state q_1 can reach h_2 from h performing one step and that are not already in $Q_P(h_2) \cup Q_N(h_2)$. Moreover, T always updates $Q_P(h)$ by adding q_1 and $Q_N(h)$ by removing q_1 if q_1 is currently present in $Q_N(h)$.

   After that, T changes the mapping represented in C. For all pairs (f_2, q_2), where f_2 is a field in the perimeter of C and $q_2 \in Q$, such that $(q_1, h) \in$

$S(f_2, q_2)$ ($q_1 \in Q'$ – see step 2)), T deletes (q_1, h) from $S(f_2, q_2)$. An explanation of this action is as follows. State q_1 has been detected to be reachable in h, when the head of A scans g and the control unit is state q'. The pair (q_1, h) can be possibly reachable from another fields of the perimeter of C and states, however, this information is no longer relevant, since the situation when A reaches q_1 in h has been already detected, thus (q_1, h) can be removed from $S(f_2, q_2)$. This action has an impact on time complexity – T will not check the same transitions across the block borders multiply. To remove (q_1, h) from all related $S(f_2, q_2)$ means to go trough the row or column in which h is located (it depends on the fact whether h is in vertical or horizontal part of the perimeter of C) and check and modify represented values of the mapping. The deletion is done just by adding auxiliary "deleted" markers so that the original values can be restored when the procedure is done.

4) If F is false, T retrieves $\mathrm{acc}_C(g, q')$. If this value is true, the value of F is changed to be true as well. Finally, T moves the head to the top-left field of B, so that it is ready to go again through steps 1) – 4).

When all repetitions of steps 1), 2), 3) and 4) are finished, T traverses fields of B', reads states detected to be reachable, computes, in which states A can leave B from the field, stores particular values of the mapping and deletes auxiliary symbols used to represent contents of Q_N's and Q_P's. One field of B' is processed in time $O(|B'|)$, the whole perimeter in time $O(|B'|^2)$. Markers related to mappings in B_{ij}'s are also removed in time $O(|B'|^2)$. In f, a representation of $\mathrm{acc}_B(f, q)$ is recorded in constant time. Let the presented procedure be denoted by $\mathcal{P}$.

For the input pair f, q to $\mathcal{P}$, time complexity of $\mathcal{P}$ is $O(|B'|^2)$. To derive it we should realize that the number of repetitions of steps 1), 2) and 4) is bounded by the the number of pairs a state q' reachable in a field h belonging to the perimeter of one of B_{ij}'s, so it is $O(|B'|)$ (here we exclude execution of step 3) from step 2)). As for step 3), it is true that it can be repeated several times for one execution of step 2), however, during $\mathcal{P}$, T never performs the step more than once for one pair q', h – it is ensured by the deletion of values from the mapping described in step 3). It means the step is repeated $O(|B'|)$ times during $\mathcal{P}$.

Each of the four steps is performed in time $O(|B'|)$. Since the whole computation requires to execute the procedure for each $f \in B'$, $q \in Q$, i.e. $|B'| \cdot |Q|$ times, it is of time complexity $O(|B'|^3)$.

It should be clear that we can generalize the presented merging process on any constant number of neighboring blocks, i.e. if $B = \bigoplus [B_{ij}]_{c_1,c_2}$, where c_1, c_2 are constants and B_{ij}'s are blocks storing mappings, then the mapping for B can be computed in time $O(|B'|^3)$.

Now, let B be a block of size $r \times s$, where $\max(r, s) \leq 2 \cdot \min(r, s) - 1$ and $\min(r, s) \geq 2$. The merges can be applied gradually in iterations – starting with mappings for $r \cdot s$ blocks of size 1×1, where each of them is computed in constant time, merging them into mappings for larger and larger blocks, finally obtaining the mapping for B. To describe the process of merges more precisely, let B_{ij}'s be blocks storing mappings (produced by previous iterations) such that $B = \bigoplus [B_{ij}]_{c,d}$. Then, in the next iteration, B_{ij}'s are merged into C_{ij}'s, where $B = \bigoplus [C_{ij}]_{c_1,d_1}$, $c_1 = \lfloor c/2 \rfloor$, $d_1 = \lfloor d/2 \rfloor$. Each $C_{p,q}$ is of the form $\bigoplus [D_{ij}]_{r_1,s_1}$,

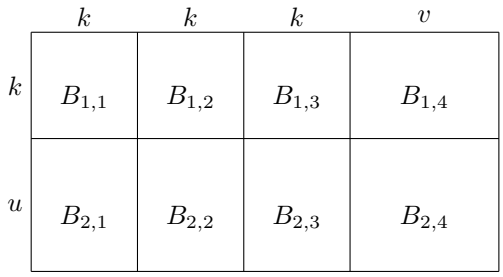


Figure 3.6: A picture divided into blocks $B_{i,j}$. The lengths u, v are between k and $2k+1$ inclusively.

where $D_{ij} = B_{2p-2+i,2q-2+j}$, $r_1 = 3$ if $p = c_1$ and c is odd, otherwise $r_1 = 2$, and, similarly, $s_1 = 3$ if $q = d_1$ and d is odd, otherwise $s_1 = 2$. In another words, mappings in four neighboring blocks are being merged together. Whenever there is a remaining odd row, resp. column of blocks, they are merged with blocks in the previous two rows, resp. columns.

Without loss of generality, let $r \geq s$. The input to the k-th iteration ($k = 1, 2, \ldots$) consists of at most $\left(\frac{r}{2^{k-1}}\right)^2$ blocks. Except the last row, resp. column of blocks, each of them is a square of size 2^{k-1}. By induction on k, we can show that the number of rows, resp. columns of any of the remaining blocks is less than or equal to $2^k - 1$. It surely holds for $k = 1$ since all blocks are of size 1×1 in this case. Performing the k-th iteration, whenever some $D = \bigoplus [D_{ij}]_{2,3}$ is obtained from D_{ij}'s, then $\mathrm{cols}(D_{1,1}) = \mathrm{cols}(D_{1,2}) = 2^{k-1}$, $\mathrm{cols}(D_{1,3}) \leq 2^k - 1$, thus $\mathrm{cols}(D) \leq 2^{k+1} - 1$. An analogous conclusion can be made for rows as well. Based on these observations, for a suitable constant c_0, an upper bound on time complexity of all iterations can be expressed as follows

$$\sum_{i=0}^{\lfloor \log_2 r \rfloor} c_0 \cdot \left(\frac{r}{2^i}\right)^2 \cdot \left(2 \cdot 2^i\right)^3 = 8 \cdot c_0 \cdot \sum_{i=0}^{\lfloor \log_2 r \rfloor} \left(\frac{r}{2^i}\right)^2 \cdot \left(2^i\right)^3 = 8 \cdot c_0 \cdot r^2 \cdot \sum_{i=0}^{\lfloor \log_2 r \rfloor} 2^i = O(r^3)$$

Now, when we have a procedure computing the mapping for a block, we can proceed with the description of the whole computation of T. Let $k = \lfloor \sqrt{n} \rfloor$. T divides P into $m_1 = \lfloor \frac{m}{k} \rfloor$ rows and $n_1 = \lfloor \frac{n}{k} \rfloor$ columns of blocks as it is shown in Figure 3.6.

Blocks are denoted by B_{ij}, each of them has the width, resp. height between k and $2 \cdot k + 1$ inclusively. T computes k in time $O(n^{\frac{3}{2}}) = O(m \cdot n)$ using the procedure given by Lemma 2 and represents it by a marker placed in the first row in the distance k from the top-left field. The division can be done by copying the distance k field by field several times so that the whole grid corresponding to borders of blocks is created. To be more precise, it is sufficient to copy k in the first row $n_1 - 1$ times. The first copy is placed after already represented k and, depending on n, the last copy need not be completed. Similarly, k is copied m_1 times into the first column. Then, all borders of blocks can be marked – copies of k determine all coordinates of the grid. The creation of one copy of k requires time k^2, a row, resp. column of the grid is marked in time n, resp. m,

thus the total time of the division is

$$O\left(n^{\frac{3}{2}} + m_1 \cdot (n_1 - 1) \cdot k^2 + m_1 \cdot n + n_1 \cdot m\right) =$$

$$O\left(m \cdot n + \tfrac{m}{k} \cdot \tfrac{n}{k} \cdot k^2 + 2 \cdot \tfrac{m \cdot n}{k}\right) = O\left(m \cdot n\right) \quad (2)$$

For each block B_{ij}, T computes the correspondent mapping. It is done in time $O(k^3)$ per a block. Since there are $m_1 \cdot n_1$ blocks, all mappings are computed in time

$$O(k^3 \cdot m_1 \cdot n_1) = O\left(k^3 \cdot \frac{m \cdot n}{k^2}\right) = O(k \cdot m \cdot n) \quad (3)$$

The final task is to find out whether there is an accepting state reachable from the initial configuration. Let the top-left corner of P be denoted by $f_{1,1}$. To decide it T performs a procedure analogous to $\mathcal{P}$. Note that it is sufficient to perform the procedure only once now, taking $f_{1,1}$ and q_0 as the input, while, in the case of merging blocks, $\mathcal{P}$ was run for each combination of a field of the perimeter and a state in Q. It is not also necessary to represent computed values of the mapping, the result depends only on $\mathrm{acc}(f_{1,1}, q_0)$.

There is a difference in number of blocks. This time, the number is not bounded by a constant. To improve time complexity of finding a non-empty set Q_N, T uses fields of the first row of P to record boolean flags indicating columns of P that contain at least one field having assigned a non-empty Q_N. To be more precise, if g is the i-th field of the first row, then the assigned flag is true if and only if the i-th column has a field f belonging to the perimeter of a block such that $Q_N(f) \neq \emptyset$. Assuming the head of T scans $f_{1,1}$, T is able to find a field with non-empty Q_N in time $O(m + n) = O(n)$. It moves the head right in the first row until it detects a flag of true value. Then it scans the marked column.

At the beginning, the only non-empty Q_N is assigned to $f_{1,1}$. Flags are initialized accordingly. During the computation, whenever T removes the last element in Q_N or adds an element to an empty Q_N, it must update the correspondent flag. In the description of $\mathcal{P}$, we had that the contribution to time complexity from one pair formed of a state reachable in a field of a perimeter is linear in the size of the perimeter. Now, the contribution must be increased by time needed to handle boolean flags, thus it is $O(n + k) = O(n)$. There are $m_1 \cdot n_1$ blocks, each of then with perimeter consisting of $O(k)$ fields. It implies, the total time complexity of the modified procedure $\mathcal{P}$ is

$$O\left(n \cdot m_1 \cdot n_1 \cdot k\right) = O(n \cdot \frac{m}{k} \cdot \frac{n}{k} \cdot k) = O\left(\frac{m \cdot n^2}{k}\right) \quad (4)$$

If we sum time complexities (2), (3) and (4), we get T computes in time

$$O\left(m \cdot n + k \cdot m \cdot n + \frac{m \cdot n^2}{k}\right) = O\left(k \cdot m \cdot n + \frac{m \cdot n^2}{k}\right) = \left(m \cdot n \cdot \sqrt{n}\right)$$

As the last remark on the proof, we show that the chosen value of k is optimal. Expression (3) grows in k while expression (4) falls in k, furthermore, (2) is always less than or equal to (3), thus to minimize time complexity means to find k for which (3) equals to (4). We derive k as follows

$$k \cdot m \cdot n = \frac{m \cdot n^2}{k}$$

$$k^2 = n$$

$$k = \sqrt{n}$$

Since k must an integer, we took $k = \lfloor\sqrt{n}\rfloor$. □

Proposition 5 *Let $A = (Q, \Sigma, q_0, \delta, Q_A)$ be a FSA recognizing L over Σ. Then, it is possible to construct a two-dimensional deterministic bounded Turing machine T that for each picture P over Σ of size $m \times n$, where $\min(m,n) \leq \lfloor\max(m,n)^{\frac{1}{2}}\rfloor$, decides whether P is in L in time $t(m,n) = O\left(\min(m,n)^2 \cdot \max(m,n)\right)$.*

Proof. The construction of T we present shares ideas used in the proof of Proposition 4. T starts by checking which of the numbers $\text{rows}(P)$, $\text{cols}(P)$ is greater. Without loss of generality, let $m \leq n$. Then, the following computation consists of two phases. During the first phase, T divides P into blocks and computes mappings for them. In the second phase, mappings in the blocks are being merged. That servers to find out if some state in Q_A is reachable.

As for the first phase, P is divided into blocks $B_1, B_2, \ldots, B_k$, where $k = \lfloor\frac{n}{m}\rfloor$. B_k is of size $m \times (m + n \bmod m)$. The other blocks are squares of size m. The division can be done in time $O(n)$ as follows. T starts scanning the top-left field of P. It marks fields of the first column as the left border of B_1 ending with the head placed on the bottom-left field. Then, it moves the head diagonally up-right, until it encounters the background symbol #. Now, it can mark the right border of B_1 and the left border of B_2, etc. When the division is done, for each block, T computes the mapping. Since A can cross between two blocks in the horizontal direction only, it is sufficient to work with mappings defined on fields of the first and last column of a block only, but this does not brings any significant improvement of time complexity, since a mapping is still computed in time $O(m^3)$ per one block, thus to compute all mappings requires time

$$O(m^3 \cdot k) = O\left(m^3 \cdot \frac{n}{m}\right) = O(m^2 \cdot n)$$

For $i = 1, \ldots, k$, let B'_i be the concatenation of the first i blocks B_i, i.e. $B'_i = B_1 \circledcirc \ldots \circledcirc B_i$. Moreover, for an arbitrary block B, let $B(i)$ denote the i-th column of B. The second phase of the computation consists of $k-1$ iterations. Each j-th iteration computes the mapping for B'_{j+1}. We summarize information that a mapping should provide (this time, the possibility to restrict the definition of a mapping on border columns only plays a significant role).

- If A enters B'_{j+1} from right in some field f and state q, we want to know, in which states and fields A can leave and, furthermore, if A can accept without leaving B'_{j+1}. Note that this information is not relevant for the last block B'_k, since A cannot enter it from right.

- If A starts in the initial configuration, we want to know, if A can accept without leaving B'_{j+1} and in which fields and states A can leave the block.

The mapping for B'_{j+1} will be stored in the B_{j+1} part only – the last column is the only border where the head of A can leave B_{j+1}, thus a square of $m \times m$ fields is a sufficient space. As for the field $B'_{j+1}(1,1)$, we use the space assigned

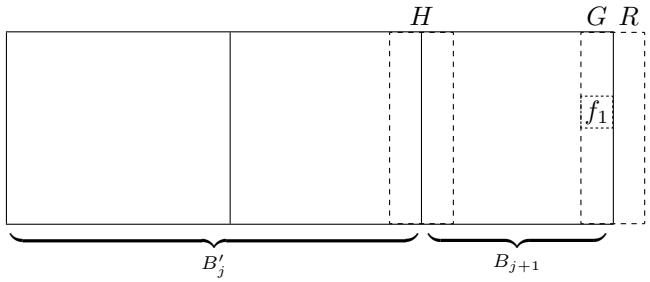


Figure 3.7: Blocks B'_j, B_{j+1} need to be merged to obtain B'_{j+1}. Field f_1 is one of the fields for which T performs the procedure – the result are states and fields in which A can leave B_{j+1} by the assumption it has entered it at the field f_1, being in state q_1.

to $B_{j+1}(1,1)$ in B_{j+1} to store values related to $B'_{j+1}(1,1)$ (see the proof of Proposition 4 for a detailed description of how the storage is organized).

At the beginning, we can consider the mapping for B'_1 to be represented, since $B'_1 = B_1$. Let us assume the mapping has been computed for B'_j $(j < k)$. We would like to merge the mappings in B'_j and B_{j+1} to compute the mapping for B'_{j+1}. To do it, T works with sets Q_P and Q_N. Let H be the set consisting of fields in the last column of B'_j and first column of B_{j+1}, G be the last column of B_{j+1}, R the first column of B_{j+2} (R is defined only if $j < k+1$). See Figure 3.7 for a scheme related to these areas. For each pair (f_1, q_1), where $q_1 \in Q$ and $f_1 \in G$, T computes as follows

1) Q_N and Q_P are initialized for every $f \in G \cup H \cup \{f_1\}$, $Q_N(f_1)$ by $\{q_1\}$, the other sets by $\emptyset$.

2) T repeats: It finds a field $f \in H$ (not in R) such that $Q_N(f) \neq \emptyset$, chooses a state $q \in Q_N(f)$ and goes through all values of $S(f,q)$. If $(f', q') \in S(f,q)$, T moves the head to f' and checks in which states A (being in the state q') can leave the block, updates sets Q_N, Q_P accordingly and modifies the mapping (see the proof of Proposition 4 for details on these actions). Note that there are three possible combinations of a direction and border, where the head of A can leave a block we are interested in: from B_{j+1} to B_j, from B_j to B_{j+1} and from B_{j+1} to B_{j+2}.

   B'_k is handled in a special way. In this case acc has to be computed only. The set R is not defined, since no movements to the right (outside B_k) can occur.

   This point is repeated until no suitable $f \in H$ can be found.

3) By checking states in sets Q_N in R, T computes and represents the mapping. The value of $\mathrm{acc}(f_1, q_1)$ is recorded and updated in states of T. At the end, all used auxiliary markers are removed and T is ready to run the routine on the next pair (field, state).

The same process is applied for the computation of values related to $B_1(1,1)$ and q_0. As we have already noted, the only difference is that $B_1(1,1)$ is associated with $B_j(1,1)$, meaning that the initialization of Q_N takes place in $B_j(1,1)$

and the computed values of the mapping are recorded into space which is reserved for $B_{j+1}(1,1)$.

Values for one pair consisting of a field and state are computed in time $O(m^2)$. There are $|Q| \cdot m + 1$ pairs to be processed, thus one merge is done in time $O(m^3)$. T has to perform $k-1$ merges to get B'_k which implies that all these merges are done in time

$$O\left(m^3 \cdot k\right) = O\left(m^3 \cdot \frac{n}{m}\right) = O\left(m^2 \cdot n\right)$$

Thus, the whole computation is of time complexity $O(m^2 \cdot n)$. □

We can put the two previous propositions together as follows.

Theorem 3 *Let $L \in L(FSA)$. There is a two-dimensional bounded deterministic Turing machine T accepting L in time $t(m,n) = O\left(m \cdot n \cdot f(m,n)\right)$, where*

$$f(m,n) = \begin{cases} \sqrt{\max(m,n)} & \text{if } \min(m,n) \geq \sqrt{\max(m,n)} \\ \min(m,n) & \text{otherwise} \end{cases}$$

Proof. T can be constructed as follows. Let P an input to T of size $m \times n$. The procedure described in Lemma 3 is used to detect whether $\min(m,n) \geq \max(m,n)^{\frac{1}{2}}$. Depending on this, the simulation from Proposition 4 or Proposition 5 is applied. □

Chapter 4

Time Complexity in Two Dimensions

4.1 NP_{2d}-completeness

The purpose of this section is to demonstrate a kind of a relationship between computations over one-dimensional strings and two-dimensional pictures. We show that pictures can be encoded by strings so that each Turing machine working over pictures can be simulated with a polynomial slowdown by a one-dimensional Turing machine working over the encoded values. We focus on the question what is the order of the polynomial, since the pure result that a simulation can be done is no surprise.

We also define classes of picture languages analogous to P, NP and NPC. Applying the simulation, we show that a picture language is NP-complete if and only if its encoded string variant is also NP-complete. It is again a simple result that we would have expected, but it give us a formal justification to treat NP-completeness in two dimensions like in one dimension.

Let Σ be a finite alphabet, $\$ \notin \Sigma$ a special symbol. We define a function $\gamma : \Sigma^{**} \to (\Sigma \cup \{\$\})^*$ encoding a picture over Σ by a string over $\Sigma \cup \{\$\}$ as follows:

$$\gamma(O) = \$o_1\$o_2\$ \ldots \$o_m\$$$

where $O \in \Sigma^{**} \setminus \{\Lambda\}$, $m = \text{rows}(O)$ and o_i is the i-th row of O. Moreover, we put $\gamma(\Lambda) = \$\$$. Obviously, γ is an injective function.

Furthermore, we extend γ on L over Σ: $\gamma(L)$ is a one-dimensional language over $\Sigma \cup \{\$\}$ such that

$$\gamma(L) = \{\gamma(O) \mid O \in L\}$$

Lemma 4 *Let $\sum_{i=1}^{k} x_i = r$, where each x_i is a non-negative real number. Then, $\sum_{i=1}^{k} x_i^2 \leq r^2$.*

Proof. The lemma can be easily proved using the inequality

$$\left(\sum_{i=1}^{k} x_i\right)^2 \geq \sum_{i=1}^{k} x_i^2$$

□

Proposition 6 *Let T_1 be a two-dimensional TM recognizing a language L in time t_1. Then, there is a one-dimensional TM T_2 recognizing $\gamma(L)$ in time t_2 such that*

$$t_2(n) = \begin{cases} O\left(t_1^4(n)\right) & \text{if } t_1 = \Omega(\text{lin}) \\ O\left(n^3 \cdot t_1(n)\right) & \text{if } t_1 = O(\text{lin}) \end{cases}$$

Moreover, whenever T_1 is deterministic, T_2 is deterministic as well.

Proof. We describe the computation of T_2. Let L be a language over some alphabet Σ. Let us consider an input string v of the form $\$w_1\$ \dots \$w_k\$$, where each w_i is a string over Σ. Note that T_2 can easily check if v starts and ends with \$. If not, v is rejected immediately. Next, let $|v| = n$. The computation consists of two phases. In the first phase, T_2 checks if $|w_1| = |w_2| = \ldots = |w_k| = l$. If not all lengths of w_i's are equal, w_i's cannot be rows of a picture over Σ, thus such an input has to be rejected. After the verification, T_2 continues by a simulation of T_1.

Let $w_1 = a_1 \dots a_r$, $w_2 = b_1 \dots b_s$, where $a_i, b_i \in \Sigma$. We denote by A_i, resp. B_i the cell of the tape containing the i-th symbol of w_1, resp. w_2. T_2 verifies $|w_1| = |w_2|$ using the procedure resembling the process of copying a block field by field:

- The initialization consists of moving the head to A_1.
- For $i = 1, \dots, \min(r, s)$, T_2 repeats: Let the head scan A_i. T_2 marks A_i by 'counted', keeps moving the head right until the first non-marked field of w_2 (i.e. B_i) is detected. T_2 marks B_i by 'counted' and returns back to the leftmost non-marked field of w_1, i.e. A_{i+1}.
- $|w_1| = |w_2|$ iff both fields marked in the last step correspond to the last symbol of w_1, resp. w_2.

When T_2 finishes checking the pair w_1, w_2, it clears all used markers and continues with the next pair (i.e. w_2, w_3), etc. Time complexity of one $|w_i| = |w_{i+i}|$ check is bounded by $c_1 \cdot |w_i|^2 + c_2$, where c_1, c_2 are suitable constants (independent on inputs). It means the comparison of all lengths is done in time

$$\sum_{i=1}^{k-1} c_1 \cdot |w_i|^2 + c_2 = c_1 \cdot \left(\sum_{i=1}^{k-1} |w_i|^2 \right) + c_2 \leq c_1 \cdot \left(\sum_{i=1}^{k-1} |w_i| \right)^2 + c_2 = O\left(n^2\right)$$

Note that we have used Lemma 4 to derive the estimation.

Let us assume now that $|w_1| = |w_2| = \ldots = |w_k|$. We show how T_2 simulates the computation of T_1 on the input $O = \bigoplus [P_{ij}]_{k,1}$, where $P_{i,1} = w_i$. During the simulation, the content of the tape after finishing a step is of the form $\$v_1\$v_2\$ \dots \$v_c\$$, where $|v_1| = |v_2| = \ldots = |v_c|$, $c \geq k$, $v_i \in \Gamma^*$, where Γ is the working alphabet of T_1. The picture $\bigoplus [P_{ij}]_{c,1}$, where $P_{i,1} = v_i$, represents a rectangular area that is a part of the working tape of T_2 that contains (in a given configuration) all fields storing symbols different to #. Note that the mentioned area need not be necessary the smallest one possible. In addition, the position of the head of T_2 corresponds to the position of the head of T_1. It is placed

over the field currently representing the field scanned by T_1. Also note that the input $\$w_1\$ \dots \$w_k\$$ represents the simulated tape in the initial configuration.

A description of how one computational step of T_1 is simulated follows: Let the head of T_1 scan the field in the i-th row, j-th column. The head of T_2 is placed over the j-th symbol of v_i. The easiest situation is when T_1 does not move the head – T_2 does not move its head too and just performs the same rewriting as T_1. Next, to proceed, let us consider a movement of the head of T_1 during its t-th step 1) in a horizontal direction, 2) in a vertical direction. Let $r(t)$, $s(t)$ denote the number of rows, resp. columns of the tape of T_1 that are represented in the tape of T_2 when the simulation of the first t steps is done.

Ad 1) If T_1 moves the head left, resp. right and $1 < j$, resp. $j < |v_i|$, then T_2 performs exactly the same rewriting and head movement as T_1. If $j = 1$ and T_2 moves the head left, first of all, T_1 appends one symbol # to the left end of each v_i, it means $\$v_1\$ \dots \$v_c\$$ is changed to $\$\#v_1\$ \dots \$\#v_c\$$. T_2 appends the symbols one by one, starting with v_1 – it moves all the remaining symbols on the tape right by one position (time proportional to $r(t) \cdot s(t)$ is required to do it). After that, T_2 marks the end of v_1 as 'processed' and continues with v_2, etc. Choosing a suitable constant c_3, we can derive an upper bound on time complexity of the whole process as follows:

$$c_3 \cdot s(t) \cdot r^2(t)$$

The case when $j = |w|$ and T_1 moves the head right is handled analogously. This time, $\$v_1\$ \dots \$v_c\$$ is changed to $\$v_1\#\$ \dots \$v_c\#\$$.

Ad 2) If T_1 moves the head down, resp. up and $i < c$, resp. $i > 1$, then T_2 needs to move its head to the j-th symbol of v_{i+1}, resp. v_{i-1}. It detects the needed j-th symbol by counting one by one the symbols precessing the j-th symbol in v_i, marking them and marking correspondent symbols in v_{i+1} (resp. v_{i-1}). After finding the desired position, T_2 clears all used markers. The position is found in time $O(|v_i|^2)$. Next, if $i = 1$ and the head of T_1 moves up, T_1 changes $\$v_1\$ \dots \$v_c\$$ to $\$\#^{|v_1|}\$v_1\$ \dots \$v_c\$$, it means it inserts a new row before the first one. The length of the inserted row is equal to the length of v_1, thus the row is created by counting the symbols of v_1 one by one. It is done in time at most

$$c_3 \cdot s^2(t)$$

Note that we consider that c_3 is chosen large enough to be applicable in this estimate as well as in the previous one.

Finally, if $i = c$ and T_1 moves the head down, T_2 expands $\$v_1\$ \dots \$v_c\$$ to $\$v_1\$ \dots \$v_c\$\#^{|v_c|}\$$. Time complexity is the same as in the previous case.

To make the derivations that appear in the rest of the proof more readable, we omit the constant c_3 – it is only a multiplicative factor. Its presence does not influent the inequalities and upper bounds we will derive.

Now, we derive time complexity of the whole simulation. We start by an observation. For some time-step t, let $r_0 = r(t)$, resp. $s_0 = s(t)$ be the number of represented rows, resp. columns of T_1's tape. Let the next two steps of T_1 (i.e. steps $t+1$ and $t+2$) be a movement in the vertical direction followed by a movement in the horizontal direction. The simulation of the first step requires time s_0^2 and of the second step time $s_0 \cdot (r_0 + 1)^2$ maximally (the first step of these two can increase the number of represented rows by one). Furthermore, let

us consider the reversed order of these simulated steps – a horizontal movement let be done first, a vertical second. In this case the first step is simulated in time $s_0 \cdot r_0^2$, the second in time $(s_0+1)^2$ maximally. Let us examine the difference in number of steps between these two estimates:

$$[s_0^2 + s_0 \cdot (r_0+1)^2] - [s_0 \cdot r_0^2 + (s_0+1)^2] =$$

$$s_0^2 + s_0 \cdot r_0^2 + s_0 \cdot 2 \cdot r_0 + s_0 - s_0 \cdot r_0^2 - s_0^2 - 2 \cdot s_0 - 1 =$$

$$2s_0r_0 + s_0 - 2s_0 - 1 = 2s_0r_0 - s_0 - 1 \geq 2s_0 - s_0 - 1 \geq 0$$

It means, the upper bound on time of the first variant is always greater than or equal to the upper bound of the second variant. We can use this conclusion to derive an upper bound on the total time of the simulation. It is sufficient to consider computations of T_1 that are sequences of steps, where no horizontal step precedes a vertical step and where are not steps in which the head does not move. In another words, the sequence consists of a group of vertical steps that are followed by a group of horizontal steps (one of these groups can be possibly empty). Let us say such a sequence is of type $\mathcal{VH}$. The estimate cannot be greater if a sequence of the same length of different type is considered. To see it, we need to realize that to simulate a step in which the head does not move cannot take more steps comparing to a step in which the head moves and, furthermore, it also does not add a new row, resp. column to be represented by T_2. Secondly, if we have a sequence $\mathcal{C}$ in which a horizontal step precedes a vertical step, we can change the order of these two steps to get a sequence $\mathcal{C}'$. The estimate derived for $\mathcal{C}'$ cannot be greater than the estimate for $\mathcal{C}$. Furthermore, the swaps can be repeated until we get a sequence of type $\mathcal{VH}$.

Let us assume the computation of T_1 is of type $\mathcal{VH}$ and there are r vertical steps followed by s horizontal steps. It holds $r(0) = k$ and $s(0) = l$. Next, let us consider the $t+1$-st computational step of T_2. If the head of T_1 moves in this step vertically, then $s(t+1) = s(t)$ and $r(t+1) \leq r(t)+1$, if horizontally, then $s(t+1) \leq s(t)+1$ and $r(t+1) = r(t)$. Thus, we have $s(i) = l$ for $i = 1, \ldots, r$ and $s(i) \leq l+i-r$; $r(i) \leq k+r$ for $i = r+1, \ldots, r+s$. Moreover, $r+s \leq t_1(n)$. An upper bound on time complexity can be derived as follows:

$$\sum_{i=1}^{r} s^2(i) + \sum_{i=r+1}^{r+s} s(i) \cdot r^2(i) \leq$$

$$l^2 \cdot r + \sum_{i=1}^{s} (l+i) \cdot (k+r)^2 =$$

$$l^2 \cdot r + (k+r)^2 \cdot (l \cdot s + \sum_{i=1}^{s} i) \leq$$

$$l^2 \cdot r + (k+r)^2 \cdot (l \cdot s + s^2) =$$

$$l^2r + k^2ls + k^2s^2 + 2krls + 2krs^2 + r^2ls + r^2s^2 \leq$$

$$(l^2 + k^2l) \cdot t_1(n) + (k^2 + 2kl) \cdot t_1^2(n) + (2k+l) \cdot t_1^3(n) + t_1^4(n) \leq$$

$$n^3 t_1(n) + n^2 t_1^2(n) + (2n+1) t_1^3(n) + t_1^4(n) \qquad (1)$$

If $t_1 = \Omega(\text{lin})$, the expression (1) is $O(t_1^4(n))$, if $t_1 = O(\text{lin})$, it is $O(n^3 \cdot t_1^4(n))$. To complete the proof, it remains to remark that if T_1 is deterministic, then the constructed T_2 is deterministic as well. □

Definition 6 *Let P_{2d}, resp. NP_{2d} denote the class of picture languages recognizable by a deterministic, resp. non-deterministic two-dimensional Turing machine in polynomial time. Furthermore, let NPC_{2d} be the class of NP_{2d}-complete languages, i.e. $L \in NPC_{2d}$ iff $L \in NP_{2d}$ and, for each $L' \in NP_{2d}$, there is a function $f : L' \to L$ such that $O \in L \Leftrightarrow f(O) \in L'$ computable by a deterministic two-dimensional TM in polynomial time.*

As usual, we call f a *polynomial transformation* and write LfL' to denote that f transforms L to L'.

Theorem 4 $P = NP \Leftrightarrow P_{2d} = NP_{2d}$

Proof. ($\Rightarrow$) Let $P = NP$, L be an arbitrary language in NP_{2d}. We want to show $L \in P_{2d}$. By Proposition 6, $\gamma(L) \in NP$, thus $\gamma(L) \in P$. $P \subset P_{2d}$, hence $\gamma(L) \in P_{2d}$. The transformation γ can be computed deterministically in polynomial time, thus $L \in P_{2d}$.

($\Leftarrow$) Let $P_{2d} = NP_{2d}$. $NP \subseteq NP_{2d} = P_{2d}$ implies $NP \subseteq P$. □

Theorem 5 *For every picture language L over an alphabet Σ, $L \in NPC_{2d} \Leftrightarrow \gamma(L) \in NPC$.*

Proof. ($\Rightarrow$) Let $L \in NPC_{2d}$. Then, by Proposition 6, $\gamma(L) \in NP$.

Let L_1 be a language in NP. Since $NP \subseteq NP_{2d}$, there is a polynomial transformation $L_1 \alpha L$ computable by a two-dimensional DTM T. Let T' be a one-dimensional DTM simulating T in polynomial time (Proposition 6). T' performs a polynomial transformation $L_1 \alpha' \gamma(L)$.

($\Leftarrow$) Let $\gamma(L) \in NPC$. There is a one-dimensional TM T recognizing $\gamma(L)$ in polynomial time. It is possible to construct a two-dimensional TM that encodes an input picture $P \in \Sigma^{**}$ to $\gamma(P)$ (in polynomial time) and then simulates T, thus $L \in NP_{2d}$.

Let L_1 be a language in NP_{2d}. By Proposition 6, we have $\gamma(L_1) \in NP$, thus there is a transformation $\gamma(L_1) \alpha_1 \gamma(L)$. It is possible to construct a function $\alpha_2 : (\Sigma \cup \{\$\})^* \to \Sigma^{**}$ computable by a two-dimensional DTM T in polynomial time as follows:

- T checks the input whether it encodes a picture over Σ. If so, T reconstructs the encoded picture and halts, otherwise it halts immediately.

Now, a transformation $L_1 \alpha L$ can be built up on the top of α_1 and α_2: A picture $P \in \Sigma^{**}$ is transformed to $\alpha_2(\alpha_1(\gamma(P)))$. □

Theorem 6 $NPC \subseteq NPC_{2d}$.

Proof. Let L be an NP-complete language over Σ. Let us assume $\$ \notin \Sigma$. If we define

$$L' = \{\$w\$ \mid w \in L\}$$

it is evident that L' is NP-complete as well. It holds $L' = \gamma(L)$. By Theorem 5, $\gamma(L) \in NPC$ implies $L \in NPC_{2d}$. □

4.2 Recognition of Palindromes

Hopcroft and Ulman present in [4] a useful tool called the *crossing sequence*. A technique based on this tool can be developed and used to show that some one-dimensional languages cannot be recognized in better than quadratic time on single-tape Turing machines. The language of palindromes is one of them. It is obvious that there is a one-dimensional (single-tape) Turing machine recognizing this language in quadratic time by comparing pairs of correspondent symbols. The existence of the mentioned lower bound implies that such an algorithm is optimal.

In the following sections, we focus on two goals:

1. We would like to generalize the crossing sequence and the technique based on it into two dimensions.

2. We study the question if the two-dimensional tape is an advantage with respect to time complexity comparing to the one-dimensional tape.

As an answer to point two, we will show that the language of palindromes can be recognized in time $O(\frac{n^2}{\log n})$ if the two-dimensional tape is used. Moreover, the generalized technique will give us the result that an algorithm of this time complexity is optimal in the case of two dimensions.

Let L_P be the language over the alphabet $\Sigma = \{a, b, c\}$ containing exactly all the palindromes of the form wcw^R, where $w \in \{a, b\}^*$.

Lemma 5 *Let Σ_1 be an alphabet. It is possible to construct a one-dimensional deterministic Turing machine, which for a non-empty input string w over Σ_1 computes the number $k = \lfloor \log_4 |w| \rfloor + 1$ and writes it on the tape in unary (let the required configuration be $w\#^k q_f$, where q_f is a final state of the machine) in time $O(n \cdot \log n)$.*

Proof. Let the input string be w, $|w| = n \geq 1$. We denote the machine we construct by T.

T uses two special markers during its computation – 'checked' and 'counter'. 'counter' marker is used to denote a distance from the right end of the input string to the right. We interpret the distance as an unary value. The value equals k at the end of the computation. T initializes the unary counter to be zero by placing the marker at the last symbol of w. Then, it works in cycles. During one cycle T moves the head from the left end of w to the right end and marks by 'checked' all not yet marked fields except every fourth field. In the initial configuration, no field is marked by 'checked'. After T reaches the right end, it increases the unary counter by one, moves the head back to the leftmost field of w and continues by the next cycle. If T detects that all fields of w have

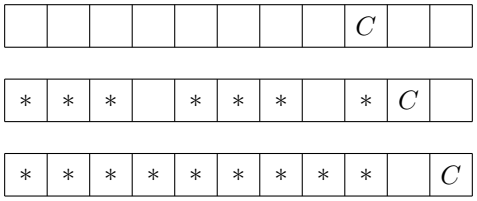


Figure 4.1: Positions of markers on the tape after finishing the initialization, the first cycle and the second (final) cycle. The input string is of length 9. $*$ represents 'checked' marker, C 'counter' marker.

been already marked, the computation is finished – T removes all used 'checked' markers, moves the head to the field containing 'counter' marker, removes it and halts. Figure 4.2 demonstrates an example of the computation over w of length 9.

It remains to prove that the number of cycles T has done equals the desired value. The field which is marked first in the i-th cycle ($i = 1, 2, \ldots$) is of index 4^{i-1}. It means, the largest possible i satisfying $4^{i-1} \leq n$ is the number of the last cycle and it is equal to $\lfloor \log_4 n \rfloor + 1$. Each cycle as well as the initialization part and the removal of used markers requires linear time, so the whole computation is done in time $O(n \cdot \log_4 n) = O(n \cdot \log n)$. □

Let us define functions ord : $\{a, b\} \to \{0, 1\}$, where $\text{ord}(a) = 0$, $\text{ord}(b) = 1$ and code : $\{a, b\}^+ \to \mathbb{N}$, where for $w \in \{a, b\}^+$, $w = x_1 \ldots x_n$:

$$\text{code}(w) = \sum_{i=1}^{n} \text{ord}(x_i) \cdot 2^{i-1}$$

The function code assigns a number (a code) to each string in $\{a, b\}^+$, this code corresponds to the binary value which a string over a two-symbol alphabet represents (in the reversed order). It is evident that for two different strings w_1, w_2 of the same length $\text{code}(w_1)$ and $\text{code}(w_2)$ are different values.

Theorem 7 *There is a two-dimensional deterministic Turing machine which recognizes the language L_P in time $O(n^2/\log n)$.*

Proof. The basic idea of the computation is to divide the whole input string into consecutive blocks of some suitable length k, code their content in unary by a marker placed in the vertical direction and compare codes of correspondent pairs (by the assumption contents of the second half of blocks is coded in the reversed order). We put k to be equal to the value from Lemma 5. Our goal will be to show that using this value it is possible to code and compare one pair of blocks in linear time. Let T denote the machine we construct.

We will work with two types of blocks - horizontal and vertical. By a horizontal block, we mean each block consisting of one column. Similarly, a vertical block is a block consisting of one row. It is sufficient to represent such a block by two different markers – the first one placed in the first field and the second one placed in the last field of the block.

T starts the computation by a verification whether the input is a picture consisting of one row (a string). If so, let w be the string and $n = |w|$. T checks whether w contains exactly one symbol c. After that, T moves the head to the first field of w and computes value k from Lemma 5. When this computation is done, T marks the position where the head has ended. It continues by dividing w into blocks. The first block is marked after copying the distance of the counter from the end of w (value k) field by field. The second block is marked copying the first block field by field, etc.

When T detects that the block it is currently marking contains the symbol c, it finishes marking blocks in the direction from left to right, moves the head back to the counter corresponding to value k and marks the last block of w copying distance k field by field (now the direction of copying is reversed, i.e. it is from right to left). After the last block of w is marked, T continues by marking next blocks from right to left until the symbol c is detected again. As a result of this process, w is divided into a sequence of blocks of length k except one block containing c. Its length is less than or equal to $2 \cdot k - 1$. We denote this block by D. If w is in L_P, then D is the central block – the created sequence of blocks can be written as $B_1, B_2, \ldots, B_m, D, \overline{B}_m, \ldots, \overline{B}_2, \overline{B}_1$. If w is not a palindrome, D need not to be the central element of the sequence. In this case, this fact will be detected during next phases of the computation, which will lead to a rejection of w, so, without loss of generality, we can assume in the following text that D is really the central block.

To compute k requires time $O(n \cdot \log n)$, to create blocks time $O(\frac{n}{k} \cdot k^2 + k \cdot n)$ – one block of length k is copied field by field to distance n and maximally $\frac{n}{k}$ blocks of length k are copied field by field to distance k – it is $O(n \cdot \log n)$ again.

During the main phase of the computation T repeatedly codes pairs of blocks and compares the unary values. Let us consider one of the pairs, e.g. B_1, $\overline{B}_1$, let B_1 contain string u. T codes the content of B_1 in k iterations. The i-th iteration consists of a creation of horizontal blocks C_i and U_i, both starting in the leftmost input field and continuing downwards (these blocks are not disjunct, they share some fields). U_i is of length 2^i and length of C_i is of length code(v), where v is the prefix of w of length i. code(v) can be possibly 0 – in this case T does not represent block C_i in the tape, but it stores this information in states. U_1 and C_1 can be marked in constant time. Let us assume that U_i, C_i have already been created. T starts the $i+1$-st iteration by scanning the $i+1$-st symbol of u and stores it in states. After that it moves the head to the fist filed of B_1 and creates U_{i+1} by doubling U_i. It is done in time at most $|U_i|^2 = (2^i)^2$ copying field by field. If the $i+1$-st symbol of u is a, then $C_{i+1} = C_i$, else C_{i+1} is the concatenation of C_i and (a copy of) U_{i+1} placed after C_i. It means, to create C_{i+1} requires time at most $|U_{i+1}|^2 = (2^{(i+1)})^2$, since $|C_i| \leq |U_{i+1}|$. When U_{i+1} and C_{i+1} are marked, T can remove markers used to represent U_i and C_i, they are no longer needed. When the k-th iteration is finished, C_k is an unary representation of code(u). Figure 4.2 demonstrates how blocks U_i and C_i are marked.

We can see that the i-th iteration is done in time bounded by $c_1 \cdot 4^i$ for a suitable constant c_1. After summing steps required by particular iterations we get

$$\sum_{i=1}^{k} c_1 \cdot 4^i \leq c_1 \cdot 4^{k+2} = O(n)$$

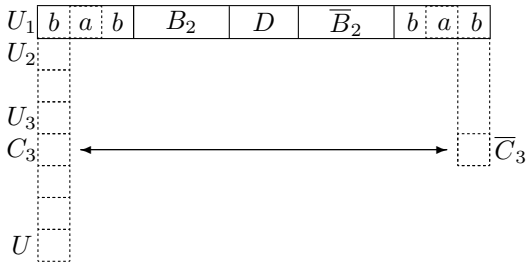


Figure 4.2: Example of an input string divided into five blocks, the first and the last block contain string *bab*. Symbols denoting vertical blocks are placed next to the last field of the block. Note, that $C_1 = C_2 = U_1$. C_3 and $\overline{C}_3$ are of length 5, which equals code(*bab*).

T continues by doubling U_k and marks all fields of the created block (U) as 'border'. It holds that code(u') $\leq |U|$ for any u' of length k. At this stage, T has done all encoding related to B_1, thus it moves the head to the rightmost field of $\overline{B}_1$ and encodes its content. Let B_1 contain $\overline{u}$. The only difference comparing to B_1 is that the reversion of $\overline{u}$ is coded instead of $\overline{u}$, i.e. an unary representation of $\overline{u}^R$ has to be computed (we want to check whether $u = \overline{u}^R$, meaning the strings can be a prefix and a suffix of some palindrome). T finishes this phase, when the block representing the desired code is computed (let the block be denoted by $\overline{C}_k$). Now, it is not necessary to continue with marking an equivalent to border block U. Instead of it, T checks if code(u) equals code($\overline{u}^R$). It moves the head to the last field of $\overline{C}_k$, then keeps moving left until it detects a field belonging to the border block U. T checks if the field is marked as the last field of C_k. If so it means the unary codes are identical and thus $u = \overline{u}^R$. In this case T marks B and $\overline{B}$ as 'verified' and continues by checking the next pair of blocks (B_2, $\overline{B}_2$). Otherwise T rejects w.

When T reaches block D, it verifies if D is the last block not marked by 'verified' yet (in another words: if D is really the central block) and if D contains a palindrome. It can be done in $O(k^2)$ time comparing pairs of fields that have to match. If the verification passes, T accepts w, otherwise it rejects it.

We can see that the coding and the comparison of one pair is done in time linear in n. The number of blocks is $\lceil n/k \rceil$, thus the whole computation requires time $O(n^2/\log n)$ (note that we have already estimated that the phase computing k and dividing w into blocks requires time $O(n \log n)$). □

4.3 Crossing Sequences

The next definition is a two-dimensional generalization of the crossing sequence defined in [4].

Definition 7 *Let $T = (Q, \Sigma, \Gamma, \delta, q_0, Q_A)$ be a two-dimensional deterministic Turing machine, P an input picture, i an integer, and let the computation of T over P be finite. The* crossing sequence of T at position i for P *is the sequence $\mathcal{K}$ constructed using the following algorithm:*

1) *$\mathcal{K}$ is initialized to be the empty sequence, j is an integral variable initialized by 1. The algorithm goes through all computational steps of T.*

2) *During the execution of the j-th step, if the control unit enters state q and the head moves from the i-th column to the $i+1$-st column or vice versa in the row of index r, the element (q,r) is appended to $\mathcal{K}$. Otherwise, if the head does not move in this way, nothing is appended to $\mathcal{K}$ at all. If j is not the number of the last computational step, it is increased by one and point 2) is repeated. Otherwise, $\mathcal{K}$ has been constructed.*

The crossing sequence at position i for an input picture P records all movements of the head across the border between the i-th and $i+1$-st column.

Note that another possibility was to define crossing sequences based on movements of the head between two neighboring rows. We can speak about vertical and horizontal crossing sequences then. However, for our purposes, it is sufficient to consider the crossing sequences we have defined only.

Let i be the index of some column. We can divide the tape into two parts – the left part, which includes every field with x-coordinate less or equal to i and the right part, which includes the remaining fields. If we know the crossing sequence at position i and the content of one of the two parts in the initial configuration, we can determine how this part will change during the computation without knowledge of the content of the other part. For example, let the content of the left part be known. Then, each time the head enters the left part (or starts there), the entry point and the current state of the control unit is given by the crossing sequence, thus it is possible to simulate the computation until the head leaves the left part again. This observation is used in the following lemma.

Proposition 7 *Let T be a two-dimensional deterministic Turing machine. Let us assume T accepts two pictures:*

$$W_1 = A_1 \oplus A_2 \oplus \ldots \oplus A_n$$

and

$$W_2 = B_1 \oplus B_2 \oplus \ldots \oplus B_m$$

where each A_p, resp. B_q is a one-column picture. Let us further assume there are two integers i, j such that $1 \leq i \leq n$, $1 \leq j \leq m$, and the crossing sequence of T at position i for W_1 is identical to the crossing sequence of T at position j for W_2. Then T accepts

$$W_3 = A_1 \oplus \ldots \oplus A_i \oplus B_{j+1} \oplus \ldots \oplus B_m$$

Proof. Let $\tau(\mathit{left}, P, k)$, resp. $\tau(\mathit{right}, P, k)$ be the portion of the tape that stores P as the input (i.e. the content of the tape corresponds to the initial configuration for the input P), where the portion consists of all columns with x-coordinate less or equal to, resp. greater than k. The computation of T in $\tau(\mathit{left}, W_3, i)$, resp. $\tau(\mathit{rigth}, W_3, i)$ is exactly the same as the computation of T in $\tau(\mathit{left}, W_1, i)$, resp. $\tau(\mathit{right}, W_2, j)$. In both cases, whenever T leaves the left part, it enters the right part in state q and the head is placed in the r-th row and vice versa. □

Proposition 8 *Let $T = (Q, \Sigma, \Gamma, \delta, q_0, Q_A)$ be a two-dimensional deterministic Turing machine computing in time t, P be an input picture over Σ. Then the sum of the lengths of all different non-empty crossing sequences of T for P, is less than or equal to $t(n)$.*

Proof. Inspecting the algorithm defining crossing sequences, we can see that the machine contributes to one of the considered crossing sequences during its horizontal movement with exactly one element. The number of movements is bounded by the number of computational steps. □

4.4 Lower Bounds on Time Complexity

We will use the crossing sequences and the previous lemmas to derive results on lower bounds on time complexity.

Theorem 8 *Let r be a real number greater than 1. There is not any two-dimensional deterministic Turing machine that recognizes L_P in time $O(n^2/log^r n)$.*

Proof. Let $T = (Q, \Sigma, \Gamma, \delta, q_0, Q_A)$ be a two-dimensional deterministic Turing machine recognizing L_P in time t. We show that $t \notin O(n^2/log^r n)$ for every real number $r > 1$.

First of all we will make an auxiliary observation. Let us consider two strings v, w of the same length belonging to L_P that can be written as $v = v_1 v_2 c v_2^R v_1^R$ and $w = w_1 w_2 c w_2^R w_1^R$. Let us assume the crossing sequences of T between v_1, v_2 and between w_1, w_2 (i.e. at position $|v_1|$, resp. $|w_1|$) are identical, $|v_1| = |w_1|$ and $v_1 \neq w_1$. Proposition 7 implies T accepts $v_1 w_2 c w_2^R w_1^R$ too, but it is not possible because $v_1 w_2 c w_2^R w_1^R$ is not a palindrome ($v_1 \neq w_1$). It means that the mentioned crossing sequences must always be different. We denote this observation by (1).

For a given odd number $n \geq 1$, we take all strings in L_P of length n. Let L_1 denote the set consisting of all these strings, s be the number of elements in Q. For L_1, we define function $p : \mathbb{Z} \to \mathbb{N}$, where $p(i)$ is equal to the average length of a crossing sequence at position i for strings from L_1.

There are $2^{\frac{n-1}{2}}$ strings in L_1. At least one half of these strings (i.e. $2^{\frac{n-1}{2}-1}$) must have the length of the corresponding crossing sequence at position i less than or equal to $2 \cdot p(i)$ (since $p(i)$ is the arithmetical average of all lengths). We need estimate the number of all crossing sequences of length less or equal to $2 \cdot p(i)$. Each j-th element from a crossing sequence is of the form (q_j, r_j). The first component of this pair is one of s different values. Since T can reach maximally $t(n)$ different rows during its computation, the second component is one of $t(n)$ different values. It means the number of all different crossing sequences of length k is maximally $(s \cdot t(n))^k$. To estimate the number of all crossing sequences of length less or equal to $2 \cdot p(i)$ we use the formula for the sum of elements of a geometric sequence:

$$\sum_{k=1}^{m} q^k = q \cdot \frac{q^m - 1}{q - 1} \leq q^{m+1}$$

where the inequality surely holds for every $q \geq 2$. In our case, we have $q = s \cdot t(n)$ and $m = 2 \cdot p(i)$, so we get that the number of required sequences is at most $(s \cdot t(n))^{2 \cdot p(i)+1}$.

Let i be an integer in $\{1, \ldots, \frac{n-1}{2}\}$. Using two previous paragraphs, we get there are at least

$$\frac{2^{\frac{n-1}{2}-1}}{(s \cdot t(n))^{2 \cdot p(i)+1}}$$

strings with identical crossing sequences at position i. By observation (1), all these strings have identical prefixes of length i. There are $2^{\frac{n-1}{2}-i}$ such different strings. It implies the inequality:

$$\frac{2^{\frac{n-1}{2}-1}}{(s \cdot t(n))^{2 \cdot p(i)+1}} \leq 2^{\frac{n-1}{2}-i}$$

We derive

$$\frac{2^{\frac{n-1}{2}-1}}{2^{\frac{n-1}{2}-i}} \leq (s \cdot t(n))^{2 \cdot p(i)+1}$$

$$2^{i-1} \leq (s \cdot t(n))^{2 \cdot p(i)+1}$$

$$i - 1 \leq (2 \cdot p(i) + 1) \cdot \log_2(s \cdot t(n))$$

We sum these inequalities for each $i \in \{1, \ldots, \frac{n-1}{2}\}$

$$\sum_{i=1}^{\frac{n-1}{2}} (i-1) \leq \sum_{i=1}^{\frac{n-1}{2}} (2 \cdot p(i) + 1) \cdot \log_2(s \cdot t(n))$$

$$\frac{n-1}{4} \cdot \left(1 + \frac{n-1}{2}\right) - \frac{n-1}{2} \leq \left(2 \cdot \sum_{i=1}^{\frac{n-1}{2}} p(i) + \frac{n-1}{2}\right) \cdot \log_2(s \cdot t(n))$$

By Proposition 8, $\sum_{i=1}^{\frac{n-1}{2}} p(i) \leq t(n)$ which implies

$$\frac{(n-1)^2}{8} + \frac{n-1}{4} - \frac{n-1}{2} \leq \left(2 \cdot t(n) + \frac{n-1}{2}\right) \cdot \log_2(s \cdot t(n))$$

$$\frac{(n-1)^2}{8} - \frac{n-1}{4} \leq \left(2 \cdot t(n) + \frac{n-1}{2}\right) \cdot \log_2(s \cdot t(n))$$

Time complexity of T has to satisfy the derived inequality for every odd $n > 0$. Let us consider $t(n) = O(\frac{n^2}{\log^r n})$, where $r > 1$. The right-hand side of the inequality is

$$\left(2 \cdot t(n) + \frac{n-1}{2}\right) \cdot \log_2(s \cdot t(n)) = O(t(n) \cdot \log t(n)) =$$

$$= O\left(t(n) \cdot \log\left(n \cdot \frac{n}{\log^r n}\right)\right) = O\left(t(n) \cdot \log n\right) = O\left(n^2 \cdot \frac{\log n}{\log^r n}\right)$$

while the expression on the left-hand side is $\Omega(n^2)$. Thus the inequality cannot be fulfilled by given t for each required n. □

Chapter 5

Two-dimensional On-line Tessellation Automata

5.1 Properties of OTA

The two-dimensional on-line tessellation automaton (OTA) is a kind of cellular automaton. We give its formal definition first.

Definition 8 *A (non-deterministic) two-dimensional on-line tessellation automaton A is a tuple $(\Sigma, Q, q_0, Q_F, \delta)$, where*

- *Σ is an input alphabet*
- *Q is a finite set of states and it holds $\Sigma \subseteq Q$*
- *q_0 is the initial state*
- *$Q_F \subseteq Q$ is a set of accepting states*
- *$\delta : Q \times Q \times Q \to 2^Q$ is a transition function*

Let $P \in \Sigma^{**} \setminus \{\Lambda\}$ be an input to A. Informally, for this input, the automaton consists of cells forming an array of size $\text{rows}(P) \times \text{cols}(P)$. For $i \in \{1, \ldots, \text{rows}(P)\}$ and $j \in \{1, \ldots, \text{cols}(P)\}$, let $c(i, j)$ denote the cell at coordinate (i, j). Moreover, let $c(0, j)$ and $c(i, 0)$ denote fictive cells that stay in the state # during the whole computation.

In the initial configuration, each cell $c(i, j)$ is in the state $P(i, j)$. The computation has a character of a wave passing diagonally across the array. A cell changes its state exactly once. In the t-th step, the cells $c(i, j)$ such that $i + j - 1 = t$ compute, i.e. in the first step, it is the cell $c(1, 1)$ only, in the second step, $c(1, 2)$ and $c(2, 1)$, etc. When $c(i, j)$ performs a computational step, the change is driven by the current states of the top and left neighbor and the initial state of $c(i, j)$. If the cell is located in the first column, resp. row, then the left, resp. top neighbor is considered to be the fictive cell we have already mentioned. Let the initial state of $c(i, j)$ be q and let q_l, resp. q_t be the current state of the left, resp. top neighbor. Then, all possible transitions of $c(i, j)$ are given by states in $\delta(q_l, q, q_t)$. Figure 5.1 shows a scheme related to the change.

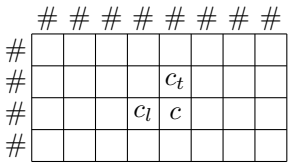


Figure 5.1: Example of an OTA working on an input of size 4×8. The transition of the cell c depends on its initial state and states of c_l and c_t. #'s denote fictive cells neighboring to cells in the first row, resp. column.

The computation consists of $\text{rows}(P) + \text{cols}(P) - 1$ steps. When it is finished, the set of states reachable by $c(\text{rows}(P), \text{cols}(P))$ determines whether A accepts P (a state in Q_F can be reached) or rejects it (otherwise).

When the input picture equals Λ, we consider OTA to be formed of one cell which of the initial state is #.

A is deterministic ($DOTA$) if $|\delta(q_1, q_2, q_3)| \leq 1$ for each triple of states q_1, q_2, q_3 in Q.

We list some of the known properties of the classes $L(OTA)$ and $L(DOTA)$ (as they can be found in [2], [5] and [12]).

1. $L(DOTA)$ is a proper subset of $L(OTA)$
2. $L(OTA)$ is closed under row and column concatenation, union and intersection
3. $L(OTA)$ is not closed under complement while $L(DOTA)$ is closed under complement
4. $L(FSA)$ is a proper subset of $L(OTA)$
5. $L(OTA)$ contains some NP_{2d}-complete problems
6. $L(DOTA)$ and $L(FSA)$ are incomparable
7. $L(DFSA)$ is a proper subset of $L(DOTA)$

In [2], the class $L(OTA)$ is considered to be a good candidate for the "ground" level of the two-dimensional theory. This opinion is justified by the fact that $L(OTA)$ has many properties we would expect the ground level class will have. It is true, that not all natural expectations are met (property 3.), however, comparing to the class $L(FSA)$, the situation is better.

Furthermore, the notion of finite-state recognizability associated with $L(OTA)$ is robust, since there are more possibilities how to characterize the class. $L(OTA)$ coincides with, e.g., the class of languages generated by *tiling systems* or the class of languages expressed by formulas of *existential monadic second order logic*.

Although the given arguments sound reasonably, there is also one that speaks against the suggestion – it is property 5. which indicates that OTA's are quite strong, since they are able to recognize some NP_{2d}-complete languages.

5.2 Simulation of Cellular Automata

In this section, we show that, in some sense, OTA's can simulate one-dimensional bounded cellular automata. As a consequence, the simulation provides a generic way how to design OTA's recognizing modifications of one-dimensional NP-complete problems recognizable by one-dimensional bounded cellular automata.

Before we start to study questions regarding the simulation, we give a definition and a brief, informal description of the model of cellular automata we work with.

Definition 9 *A one-dimensional* cellular automaton *is a tuple* $C = (Q, \delta, Q_I, Q_A)$*, where* Q *is a finite set of states,* $\delta : Q^3 \to 2^Q$ *is a transition function,* $Q_I \subseteq Q$ *is a set of initial states and* $Q_A \subseteq Q$ *is a set of accepting states. In addition, it always holds* $\# \in Q \setminus Q_I$*.*

Let us consider a cellular automaton C and let $\Sigma = Q_I \setminus \{\#\}$. Informally, C can be interpreted as a sequence (infinite in both directions) of cells, where, at any given time, each of them is in some state – an element in Q. During a computational step, each cell changes its state depending on the state and current states of two neighboring cells. If the cell is in a state q and the left, resp. right neighbor in a state q_L, resp. q_R, then all possible new states that the cell can enter are elements in $\delta(q_L, q, q_R)$. If there is no such a state, the computations halts and the input is rejected. As usual, we say C is deterministic iff δ maps every triple to a set having at most one element. Any configuration of C can be expressed by a string, where particular characters correspond to states of cells. Inputs to cellular automata are non-empty strings over Σ. If $\sigma \in \Sigma^+$ is an input, the correspondent initial configuration is $\#^\infty\sigma\#^\infty$. We say C is *bounded* if δ is $\#$-preserving, i.e. if $\# \in \delta(p, q, r)$ implies $q = \#$ and $\delta(p, \#, r) \subseteq \{\#\}$ for all $p, q, r \in Q$. If C is bounded, its configuration can be expressed using the form $\#\alpha\#$, where $\alpha \in Q^+$ and $|\alpha| = |\sigma| = n$. In this case, we can imagine that C consists of n cells $c_1, c_2, \ldots, c_n$, where each i-th cell stores i-th character of σ in the initial configuration. We consider c_1, resp. c_n to have a fictive left, resp. right neighbor staying in the state $\#$ during the whole computation. C accepts σ if the cell storing the leftmost symbol of σ (which is c_1 if we consider the described bounded automaton) can reach some state in Q_A.

In [20], there is described a procedure how to synchronize cells of a one-dimensional bounded cellular automaton by the assumption the leftmost (resp. rightmost) cell is in some distinguished state q_a while the other cells are in a passive state q_p (meaning these cells stay in this state until they are activated by a signal sent by the leftmost cell). At the end of the process, all cells enter a state q_s first time in the same step. It can be interpreted they are synchronized at this moment. The procedure requires $3 \cdot n$ computational steps maximally (n denotes the number of cells of the automaton).

Let Σ be an alphabet and \$ a special symbol not contained in Σ. We define a function $\tau : \Sigma^* \times \mathbb{N}^+ \times \mathbb{N}^+ \to (\Sigma \cup \{\$\})^{**}$, where $P = \tau(w, m, n)$ is of size $m \times n$ and, for $w = a_1 \ldots a_k$ ($a_i \in \Sigma$), it fulfills

$$P(i,j) = \begin{cases} a_j & \text{if } i = 1 \text{ and } j \leq |w| \\ \$ & \text{otherwise} \end{cases}$$

It means w is placed in the first row of P, starting in the top-left corner. The remaining fields of P contain \$.

Theorem 9 *Let $C = (Q, \Sigma, Q_A, \delta)$ be a deterministic one-dimensional bounded cellular automaton recognizing a language L_1. Let \$ $\notin \Sigma$. There is a DOTA A recognizing L_2 such that a picture P is in L_2 if and only if the following conditions are fulfilled:*

1) *P equals $\tau(w, m, n)$ for some suitable positive integers m, n and a string $w \in L_1$.*

2) *Let $t : L_1 \to \mathbb{N}$ be a function, where, for $v \in L_1$, $t(v)$ is the number of steps that C performs when computing over v. Let $k = |w|$, where w is the string from point 1). Then*

$$\text{rows}(P) \geq \frac{k+1}{2} + 3 \cdot k + t(w)$$

and

$$\text{cols}(P) \geq k + 1 + 3 \cdot k + t(w)$$

Proof. We will construct a *DOTA* A simulating C. Let P be an input to A and let $P = \tau(w, m, n)$ for some positive integers m, n and $w \in \Sigma^*$ (not necessarily in L_1), where $|w| = k \leq n$, $m \geq \frac{k+1}{2} + 3 \cdot k + t(w)$, $n \geq k + 1 + 3 \cdot k + t(w)$. The other inputs will be discussed separately in the end of the proof.

Figure 5.2 outlines the main idea of the simulation. The computation of A is divided into four phases. During the first phase, w is moved to the diagonal - it is represented in $\lceil \frac{k+1}{2} \rceil$ cells, where each of the cells stores two input characters. An exception is the cell corresponding to the end of w which can store at most one character. When the first phase is completed, it is possible to simulate one step of C during two steps of A. However, before the whole process can be launched it is necessary to synchronize cells of A so that they are able to start at the same moment. This is done during the second phase. The third phase consists of a simulation of C and finally, the fourth phase just delivers information about the result of the simulation to the bottom-right cell of A so that A can correctly accept or reject the input.

During the computation, odd and even computational steps of A are required to be distinguished. To achieve this, when $c(1, 1)$ performs its step, it is marked as a cell belonging to an odd step. In the following steps, the proper marker is assigned to a computing cell depending on the marker recorded in the left or top neighbor.

A description of the phases in more details follows. Let $w = a_1 a_2 \dots a_k$, where each $a_i \in \Sigma$. For $i = 1, \dots, m$ and $j = 1, \dots, n$, let c_{ij} denote the cell of A at coordinate (i, j). The cells $c_{1,1}, c_{2,1}, \dots, c_{k,1}$ store symbols of w at the beginning. The goal of the first phase is to represent w in the cells $d_i = c_{k+1-s+i, s-i+1}$ for $i = 1, \dots, s$ and $s = \lceil \frac{k+1}{2} \rceil$. The representation should be as follows. For each $i < s$, d_i stores $a_{2 \cdot i-1}$ and $a_{2 \cdot i}$, d_s stores a_k if k is odd. and an 'end of w' marker if k is odd, else it is marked as the right end of w only. In addition, d_1, resp. d_s is marked as the left, resp. right end of w.

To reach the desired configuration, A computes according to the following rules:

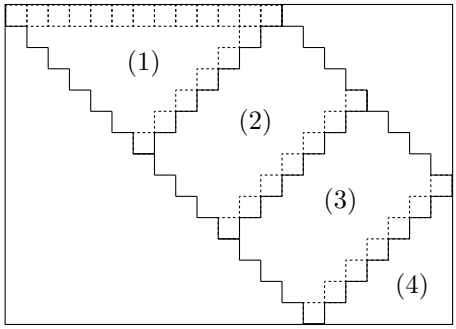


Figure 5.2: Simulation of one-dimensional cellular automaton. Groups of cells involved in one of the four computational phases are distinguished.

a)

$(a_1$	$(a_1a_2$	a_3	a_3a_4	$)\$$	$\$$
	$(a_1a_2$	$(a_1a_2$	a_3a_4		
		$(a_1a_2$			

b)

		q_5q_6	
	q_3q_4	q_5q_6 q_3q_4	r_5r_6
q_1q_2	q_3q_4 q_1q_2	r_3r_4	
	r_1r_2		

Figure 5.3: *a*) Moving of the string $a_1a_2a_3a_4a_5$ to the highlighted diagonal. Cells are labelled by symbols they represent after performing the transition. ")", resp. "(" denotes the right, resp. left end marker. *b*) Simulation of one parallel step of the cellular automaton. In the beginning, simulated cells are in states q_i, after performing the step, each q_i is changed to r_i.

- During each odd computational step of A – a cell checks if the top neighbor is in a state that represents a pair of symbols. If so, the symbols are copied to the cell. Cells in the first row do not perform any changes except the cell $c_{1,1}$, which records the presence of the left end of w. This marker is copied together with the represented symbols in all next steps.

- During each even step – a cell checks the state of its left neighbor. If the neighbor represents a pair of symbols, the cell copies them. If the cell is placed in the first row and stores a_i, then it reads a_{i-1} stored in the left neighbor and represents the pair (a_{i-1}, a_i).

- The only exception from the previous two rules is the cell $c_{1,k+1}$ which stores \$ following the last symbol of w. If $k+1$ is odd, $c_{1,k+1}$ copies a_k and marks the right end of w, else it marks the end only. After A finishes the $k+1$-st computational step, the desired diagonal representation of w is formed.

See Figure 5.2 for an illustrative example of the described process.

The second phase simulates the synchronizing procedure we have already mentioned. The rightmost cell of C, which is represented in $c_{1,k+1}$ or $c_{2,k}$, is taken as the only cell in the active state. Thanks to the synchronization, a simulation of all cells of C (working over w) can be started at the same moment (i.e. in one computational step of A). Both phases, (2) and (3), are similar – in the next paragraphs we need not to distinguish between them. Comparing to the first phase, cells of A are required to behave differently during the second and third phase, thus there must be some notification about the change. The notification is done by spreading signal 'phase one ended'. The signal is generated by the cell $c_{1,k+1}$. In the following steps, each cell which detects a neighbor marked by the signal marks itself as well. When a cell is marked, it computes as it is described in the next paragraphs. Since the signal is being spread faster than the signals related to the synchronization process (during two steps of A, signal 'phase one ended' notifies two additional cells of A simulating together 4 cells of C, while there is one step of C simulated only), this consecutive notification is sufficient. It is true that some cells belonging to the second phase will not be notified about the beginning of the second phase, however, since the cells of C simulated by them are in the passive state, they are required to copy represented states only, exactly as it happens during the first phase.

Let us assume some configuration of C is represented in a diagonal $\mathcal{D}_1$. Let $\mathcal{D}_1$ be formed of s cells c_{x_1+i,y_1-i}, where $i = 0, \ldots, s-1$ and (x_1, y_1) is the coordinate of the bottom-left cell of $\mathcal{D}_1$, A be performing two computational steps that should produce the next configuration of C. In the first step, each cell which has the left and top neighbors in $\mathcal{D}_1$ records states represented by the neighbors. The diagonal consisting of these cells is $\mathcal{D}_2 = \{c_{x_1+1+i,y_1-i} \mid i \in \{0, \ldots, s-2\}\}$. In the second step, each cell that has at least one of the neighbors in $\mathcal{D}_2$, i.e. each cell in $\mathcal{D}_3 = \{c_{x_1+1+i,y_1-1-i} \mid i \in \{0, \ldots, s-1\}\}$, simulates one computational step of represented cells of C. For example, let us consider some $c_{ij} \in \mathcal{D}_1$ storing states of two cells of C (denoted c'_1 and c'_2) at the moment when the l-th computational step of C is done. Then, the cell $c_{i+1,j-1} \in \mathcal{D}_3$ can retrieve states of c'_1, c'_2 (recorded in the top neighbor of $c_{i+1,j-1}$) as well as states of neighbors of c'_1, c'_2 in C (recorded in the left and the top neighbor), thus $c_{i+1,j-1}$ can simulate the $l+1$-st computational step of c'_1 and c'_2 and record the resulting states. A portion of cells performing the described simulation is shown in Figure 5.2.

The fourth phase is started at the moment the leftmost cell of C reaches an accepting state. If w is in L_1, it occurs after the simulation of at most $t(k)$ steps. In this case, signal 'input accepted' is started to be spread. First of all, the cell of A that detects C has accepted w is marked to carry the signal. Then, each cell of A having a left or top neighbor carrying the signal is marked as well. Using this principle, the signal reaches the bottom-right cell of A, which indicates the input should be accepted. If C does not accept, no signal is generated and A rejects. Note that, during the fourth phase, some cells of A still work as it was given in the description of the second and third phase, however, this process does not interfere with spreading of the signal, thus it has no affect on the result.

It remains to discuss, how A detects that the input cannot be written as $\tau(w, m, n)$ for some suitable values w, n, m, where $m \geq \frac{k+1}{2} + 3 \cdot k + t(w)$ and $n \geq k + 1 + 3 \cdot k + t(w)$. First of all, during the simulation, A can easily verify

if the only symbols different to \$ are placed in the first row as a contiguous sequence starting at the first field – if a cell of A contains this symbol, it checks if the top neighbor is in state # and if the left neighbor was originally (before it performed its computational step) in a state different to \$. Whenever a problem is encountered, a signal notifying bad input is generated and spread to the bottom-right cell. A verification of m and n is done automatically. If one of these values is less than it should be, then A has not a space large enough to perform all four phases, thus 'input accepted' signal cannot be generated implying the input is rejected. Note that phase one requires $\lceil \frac{k+1}{2} \rceil$ rows of cells and $k+1$ columns to be performed, phase two additional $3 \cdot k$ rows and the same number of columns and finally, phase three additional $t(w)$ rows and also $t(w)$ columns. □

The simulation we have presented can be improved in the case of non-deterministic OTA's.

Theorem 10 *Let $C = (Q, \Sigma, Q_A, \delta)$ be a non-deterministic one-dimensional bounded cellular automaton recognizing a language L_1. Let \$ $\notin \Sigma$. For $v \in L_1$, let $t(v)$ be the minimal length of a computation among all accepting computations of C when computing over v. There is a non-deterministic OTA automaton A recognizing L_2 consisting of pictures that can be written as $\tau(w, m, n)$, where $w \in L_1$, $m \geq \frac{|w|+1}{2} + t(w)$ and $n \geq |w| + 1 + t(w)$.*

Proof. We will modify the computation presented in the proof of Theorem 9. It is possible to remove the second phase (the synchronization). Using non-determinism, an equivalent configuration, where all cells are synchronized, can be guessed immediately after the representation of w is created in a diagonal. The other phases remain the same, so we describe the modification of the synchronizing process only. Let cells of A be denoted by $c_{i,j}$ again and let $r = \lceil \frac{k}{2} \rceil$.

Let each cell taking part in the first phase non-deterministically guess if it belongs to the diagonal $\mathcal{D} = \{c_{k-r+i,r+1-i} \mid i \in \{1, \ldots, r\}\}$ or not (for a cell, the guessing is performed when the cell performs its computational step). Note that cells in this diagonal compute during the k-th step of A, i.e. one step before the first phase is finished. If a cell decides it belongs to $\mathcal{D}$, it records this fact in its state (let us call such a cell to be 'synchronized'). Each cell related to the first phase checks the presence of the 'synchronized' marker in its left and top neighbor. By the position of a cell c and by its initial state, we distinguish the following cases:

1) c is placed in the first row, it is in the state \$ and this is the first \$ following the last symbol of w (i.e. $c = c_{k+1}$) – c checks if the left neighbor is 'synchronized'. If not, the cell generates 'bad synchronization' signal, that is spread to the bottom-right cell and causes that the correspondent computational branch of A does not accept.

2) c is placed in the first row and it is in some state in Σ – if the left neighbor is 'synchronized' c generates 'bad synchronization' signal.

3) c is not a cell of the first row and both its neighbors represent some symbols of w. Now, if exactly one of the neighbors is 'synchronized', then 'bad synchronization' is generated.

It should be evident that no 'bad synchronization' signal is generated if and only if all cells in $\mathcal{D}$ are the only cells that guess they are 'synchronized' – conditions 1) and 2) force c_k to be the first 'synchronized' cell in the first row, condition 3) inducts that a cell in $\mathcal{D}$ preceding a 'synchronized' cell is 'synchronized' as well.

Let us consider a computational branch of A, where the synchronization is correctly guessed. Cells computing in the $k+1$-st step verify this fact and become 'active'. From now, no more guesses related to the synchronization are needed ('active' status is spread during the remaining parts of the computation), the simulation of C can begin following the same scenario as it was described for the deterministic variant. □

Example 5 Let us consider a well known NP-complete problem – the knapsack problem. Informally, an instance of the problem is a finite sequence of positive integers $n_0, n_1, \ldots, n_k$ $(k \geq 1)$ represented in binary. These numbers are commonly interpreted as a knapsack of size n_0 and k items of sizes $n_1, \ldots, n_k$. The question is whether it is possible to select some of the available items so that they exactly fit into the knapsack. In another words, if there is a subset of indices $I \subseteq \{1, \ldots, k\}$ such that $n_0 = \sum_{i \in I} n_i$.

We show that the knapsack problem can be decided by a one-dimensional non-deterministic bounded cellular automaton in time $t(n) = n$. For our purposes, we will use a special format of inputs over the alphabet $\Sigma = \{0, 1, a, b\}$. We encode one sequence $n_0, n_1, \ldots, n_k$ by the string $w = w_0 a w_1^R b w_2^R b \ldots b w_k^R$, where w_i is n_i written in binary (w^R is the reversion of w). It means $w_i = a_{i,1} a_{i,2} \ldots a_{i,l_i}$, where $a_{i,1} = 1$, for all $j = 2, \ldots, l_i$: $a_{i,j} \in \{0, 1\}$ and

$$\sum_{j=1}^{l_i} a_{i,j} \cdot 2^{l_i - j} = n_i$$

Note that w contains exactly one symbol a. All remaining delimiters between binary values are b's.

We define L_K to be the language over Σ containing exactly all strings encoding an instance of the knapsack problem that has a solution.

Lemma 6 *There is a bounded one-dimensional cellular automaton recognizing L_K in time $t(n) = n$.*

Proof. Let us consider a well formed input of the form $w_0 a w_1^R b w_2^R b \ldots b w_k^R$. Let the cell storing a in the initial configuration be denoted by c_a.

The idea of the presented algorithm is to non-deterministically choose a subset of items and subtract their sizes from the size of the knapsack. The cells storing w_0 in the initial configuration are used during the computation to form a binary counter which is initialized by the knapsack's size and being decremented by the sizes of selected items. The cells positioned after c_a only shift the string $w_1^R b \ldots w_k^R$ left, in each step by one symbol. The last symbol of the shifted string is followed in the next step by a special signal E_1 indicating the end. The signal is generated by the rightmost cell. Since all cells preceding c_a cannot be distinguished immediately, they also shift the symbols left, however, this shifting is not of any importance for the computation. Moreover, all cells remember its initial state (this information will be needed in the cells of the counter).

The cell c_a is consecutively feeded by one symbol of $w_1^R b \dots w_k^R$ in each step. If the currently received symbol is the first symbol of some w_i^R, i.e. the least significant bit of w_i, c_a non-deterministically decides if w_i will be added to the knapsack or not. In the latter case, bits of w_i are absorbed by c_a. It means, no signals are sent to the cells on the left. In the former case, for each received bit 1, resp. 0 of w_i^R, c_a sends left signal S_1, resp. S_0 representing subtraction of 1, resp 0. Moreover, if the received bit is the least significant (i.e. the first bit of w_i^R), c_a sends together with S_i signal N announcing the beginning of a new subtraction. The idea of counting is based on delivering a signal generated by the i-th bit of w_j^R to the correspondent cell representing the i-th bit of the counter. Note that to achieve this we have decided to code all w_i, $i > 0$ in the reversed form. The last thing to be mentioned is that c_a changes E_1 to E_2 when the signal is received (the purpose of this change will be clear later).

Let us take a closer look at the behavior of cells forming the counter at the moment they receive one of the previously defined subtraction signals. Let c be one of these cells, keeping a bit d. Moreover, let f be a boolean flag which of value is stored in states of c. The flag is used to control delivering of S_i signals. If f is true, it indicates that c awaits signal S_i – when the signal is received f is changed to false meaning that next signals S_i should be passed to next cells. One more signal is still needed – signal D will be generated when a carriage from a bit to higher bits occurs during a subtraction. We have already mentioned that N can be sent together with signal S_0 or S_1. We consider N to be processed by a cell, which receives it, first (before S_i). A detailed description of how c processes received signals follows.

N) c sets f to be of true value (meaning c is the target cell for the firstly received signal S_i), N is sent to the left neighbor.

S_0) If f is false c sends S_0 to the left, else c changes f to false and, since 0 is subtracted from the bit d kept by the cell, no additional changes are needed to be performed.

S_1) Again, if f is false c sends S_1 to the left, else c changes f to false. If $d = 1$, d is changed to 0, otherwise d is changed to 1 and a signal D is generated and sent to the left neighbor (a decrement is needed in higher bits to complete the subtraction).

D) If $d = 1$, d is changed to 0, else d is changed from 0 to 1 and D is sent to the left neighbor.

If the leftmost cell is about to send S_i or D to its left neighbor, it indicates that a negative value has been reached by the counter, thus the input is rejected. While moving from c_a to the leftmost cell, signal E_2 checks if all bits of the counter are 0. If so, it means the knapsack problem has been solved successfully and the input is accepted. To verify correctness of the presented algorithm, it is sufficient to realize that signals handling subtractions of the counter never mutually interfere.

It remains to discuss how the automaton detects badly formatted inputs. If there is no symbol a contained in the input, the leftmost cell receives E_1 instead of E_2. If there are two or more symbols a, one of the correspondent cells receives E_2 instead of E_1 since E_1 is never changed to E_2. If one of the cells forming the

counter stores b at the beginning, it is detected when the cell receives some of the substraction signals or E_2. And finally, it can be checked by c_a whether each binary value representing the size of an item starts with bit 1 (to do it, c_a needs always to remember the bit received in the previous step) and the leftmost cell of the automaton checks the highest bit of the knapsack size.

The computation is of time complexity exactly $|w|$ for any $w \in \Sigma^+$ – signal E_1 is generated in the first step, after next $|w| - 2$ steps, it reaches the second cell (being possibly changed to E_2), the first cell detects it and finishes the computation in one more step. □

Proposition 9 *Let L be the following language*

$$L = \{\tau(w, m, n) \mid w \in L_K \wedge m \geq \frac{|w| + 1}{2} + |w| \wedge n \geq 2 \cdot |w| + 1\}$$

Then, L is NP_{2d}-complete and it is recognizable by a OTA.

Proof. The automaton can be constructed based on Theorem 10 and Lemma 6. NP_{2d}-completeness of L is implied by Lemma 5. □

Note that, in the literature, it is possible to find some specific NP_{2d}-complete languages recognizable by OTA. An example is the language of 3-colorable maps given in [12]. Our simulation offers a more general approach. On the other hand, NP_{2d}-complete languages like the language L in Proposition 9 are only extensions of one-dimensional languages given by the function τ – their structure is not based on the two-dimensional topology.

Chapter 6

Two-dimensional Forgetting Automata

In this chapter we present a two-dimensional generalization of forgetting automata as it is given in [1] and [9]. Forgetting automata are bounded Turing machines with restricted capabilities of rewriting symbols on the tape. They can rewrite the content of a field by the special symbol @ only (we say, they erase it). One-dimensional forgetting automata were studied for example in [7] and [8]. It has been shown how they can be used to characterize the class of one-dimensional context-free languages. Later, in section 7.3, we will extend some of these ideas and show that two-dimensional forgetting automata can recognize languages generated by grammars with productions in context-free form.

6.1 Technique Allowing to Store Information in Blocks

Definition 10 Two-dimensional forgetting automaton *is a tuple* $(Q, \Sigma, q_0, \delta, Q_F)$ *such that the tuple* $(Q, \Sigma \cup \{\#, @\}, \Sigma, q_0, \delta, Q_F)$ *is a two-dimensional, bounded Turing machine.* $@ \notin \Sigma$ *is a special symbol called the erase symbol. In addition, whenever* $(a_2, q_2, d) \in \delta(a_1, q_1)$*, then* $a_2 = a_1$ *or* $a_2 = @$.

We abbreviate a two-dimensional forgetting automaton by FA, a deterministic FA by DFA and the class of languages that can be recognized by these automata by $L(FA)$ and $L(DFA)$ respectively.

Before we start to present what forgetting automata are able to compute, we show a very useful technique of how a forgetting automaton can store and retrieve information by erasing some symbols on the tape so that the input picture can be still reconstructed. The technique comes from [10]. Two of our results regarding forgetting automata are strongly based on it.

Example 6 Let $A = (Q, \Sigma, q_0, \delta, Q_F)$ be a FA. Let $\Sigma_0 = \Sigma \setminus \{@, \#\}$, O be an input picture to A and M be the set of the tape fields containing P. Let $P(f)$ denote the symbol contained in the field $f \in M$. Then, for each $G \subseteq M$, there is $s \in \Sigma_0$ such that $|\{f \mid f \in G \wedge P(f) = s\}| \geq \left\lceil \frac{|G|}{|\Sigma_0|} \right\rceil$. If the automaton A

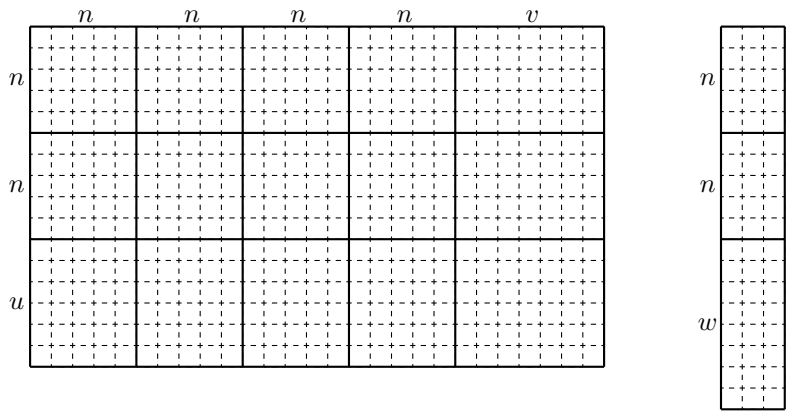


Figure 6.1: Decomposition of M into blocks of fields when both dimensions of M are at least n and when the width is lower than n. It holds that $n \leq u, v, w < 2n$.

erases some fields of G containing the symbol s, it is still able to reconstruct the original content – the erased symbol is always s. Each such field can therefore store 1 bit of information: the field is either erased or not erased. It is thus ensured that G can hold at least $\left\lceil \frac{|G|}{|\Sigma_0|} \right\rceil$ bits of information.

Let us consider M to be split into blocks of size $k \times l$, where $n \leq k < 2n$, $n \leq l < 2n$ for some n. The minimal value for n will be determined in the following paragraphs. In the case when the height, resp. width of the picture is lower than n, the blocks will be only as high, resp. wide as the picture. In the case of both the height and width of P being less than n, an automaton working over such an input can decide whether to accept it or reject it on the basis of enumeration of finitely many cases.

If both the width and height of M are at least n, all blocks contain $n \times n$ fields, except the blocks neighboring with the bottom border of M, which can be higher, and the blocks neighboring with the right border of M, which can be wider. Nevertheless, both dimensions of each block are at most $2n - 1$.

An example of a division into blocks can be seen on Figure 6.1. Let us consider each block B_i of the decomposition to be divided into two disjunct parts – F_i and G_i, where F_i consists of the first $|\Sigma_0|$ fields of B_i. We can choose the size of the blocks arbitrarily, so that a block will always contain at least $|\Sigma_0|$ fields. G_i contains the remaining fields of B_i. Let $s_r \in \Sigma_0$ be a symbol for which

$$|\{f \mid f \in G_i \wedge P(f) = s_r\}| \geq \left\lceil \frac{|G_i|}{|\Sigma_0|} \right\rceil$$

The role of F_i is to store s_r: if s_r is the r-th symbol of Σ_0 then A stores it by erasing the r-th field of F_i. Now, A is able to determine s_r, but it needs to store somewhere the information about the symbol originally stored in the erased field in F_i. A uses the first $|\Sigma_0|$ bits of information that can be stored in G_i. If the erased symbol in F_i was the t-th symbol of Σ_0 then the t-th occurrence of s_r in G_i is erased, allowing A to determine the erased symbol in F_i. This way a maximum of $|\Sigma_0|$ bits of available information storable in G_i will be lost. For

any block B_i containing m fields this method allows A to store at least

$$\left\lceil \frac{m - |\Sigma_0|}{|\Sigma_0|} \right\rceil - |\Sigma_0|$$

bits of information in B_i.

Proposition 10 *Let $A = (\Sigma, Q, Q_A, \delta)$ be an OTA and L be the language which is accepted by A. It is possible to construct a two-dimensional forgetting automaton M recognizing L. Moreover, if A is deterministic, then M is deterministic as well.*

Proof. Let P be an input to A of size $m \times n$. Each computational branch of A is fully identified by a sequence $\mathcal{C} = \{C_i\}_{i=1}^{m+n-1}$, where each C_i is a sequence of states in which the cells computing in the i-th parallel step finish. The number of the cells performing a parallel computational step is always at most $\min(m, n)$. Let $k \in \mathbb{N}^+$ be a constant used to drive some parts of the computation of M. A suitable value for k will be derived later as a consequence of all requirements on M.

If $\min(m, n) < k$, then A can be simulated even by a finite state automaton – M need not rewrite any symbols, it can operate as follows. M starts by reading the symbol in the top-left corner, simulating the first step of A and storing the obtained configuration C_1 in states. If more configurations are reachable, M non-deterministically branches for each of them. M finishes the phase by moving the head to the rightmost field of the neighboring diagonal which of cells compute in the second step. Next, let us assume M stores C_i in states (since $|C_i| < k$, it is always possible) and the head scans the rightmost field of the diagonal corresponding to C_{i+1}. To simulate the $i + 1$-st step of A, M scans and stores the initial states of cells – one by one. After that, using stored information on C_i, it is able to determine all states reachable by cells corresponding to C_{i+1}. C_{i+1} is recorded in states (this information replaces the information on C_i, which is no longer needed) and the head is moved to the rightmost field of the next diagonal. M repeats the described process until all steps of A have been simulated.

Let us focus on the case $\min(m, n) \geq k$ now. M simulates steps of A again, however, it cannot record an arbitrary configuration in states. Instead of it, it uses the technique described in Example 6 to record configurations in the tape. We consider P to be divided into blocks of sizes determined by k. To spare available space, configurations of A are not computed and stored one by one – M computes and stores configurations $C_{i \cdot 4k}$ for $i = 1, 2, \ldots$ only. A configuration is stored in blocks the diagonal of related cells goes through. Since the width an height of a block are both less than $2k$, it is evident that one block is intersected by at most one diagonal as it shows Figure 6.2. Let us list, which information a block should represent:

- One bit is used to distinguish if the block is intersected in its first row or not (if not, the diagonal of cells intersects the last column).
- A positive integer less than $2k$ determines, which field of the first row (resp. last column) is in the intersection.
- At most $2k$ integers between 1 and $|Q|$ are needed to represent states of cells corresponding to the diagonal.

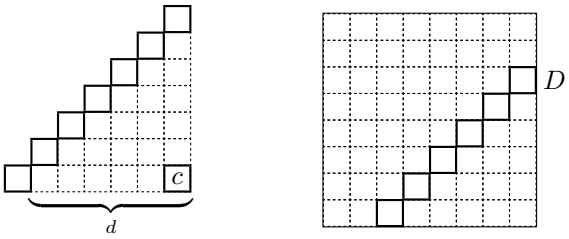


Figure 6.2: The first picture is a portion of a cellular automaton and it illustrates, which cells computing in the i-th step affect the cell c that computes in the step $i + d$. There are at most $d + 1$ such cells (the number is smaller if the distance from c to the top or left border is less than $d + 1$). The second picture shows the intersection between a block and the diagonal D corresponding to a subset of cells computing in the same step.

If all numbers are represented in binary, the capacity of

$$1 + \lfloor \log_2 2k \rfloor + 2k \cdot \lfloor \log_2 |Q| \rfloor$$

bits is always sufficient. Since the expression is $O(k)$, while a block can store $\Omega(k^2)$ bits, a suitable constant k exists.

Assuming C_i is represented in blocks, we describe how M computes and represents C_{i+4k}. Let c be a cell of A computing in the step $i + 4k$. The state c finishes in is determined by states of at most $4k + 1$ cells related to C_i (see Figure 6.2). M retrieves these states from blocks, computes the new state of c directly and represents it in a proper block. To compute C_{i+4k}, M processes all related cells, starting with the rightmost one. If C_j, where $j > m + n - 1 - 4k$, has been computed (it can be detected by checking if there are at least $4k$ next diagonals or not), after reading states in C_j, M can compute the final configuration C_{m+n-1} that consists just of one state. It should be obvious that for a deterministic automaton A the simulation is deterministic as well. □

Lemma 7 *There is a DFA recognizing the language $L = \{a^n b^n \mid n \in \mathbb{N}\}$ over $\Sigma = \{a, b\}$.*

Proof. We describe how to construct a DFA A accepting L. For an input P over Σ, A checks whether $\text{rows}(P) = 1$ and whether P is formed by two consecutive blocks: the first one consisting of a's and the second one consisting of b's (this can be checked even by a $DFSA$). After that, it repeatedly erases pairs a and b, until there is no complete pair of these symbols. At this stage, if all symbols has been erased, P is accepted, otherwise it is rejected. □

Theorem 11

- *$L(OTA)$ is a proper subset of $L(FA)$*
- *$L(DOTA)$, $L(DFSA)$ and $L(FSA)$ are proper subsets of $L(DFA)$*

Proof. Let L be the language from Lemma 7. Since FSA behaves like a one-dimensional two-way finite-state automaton and OTA like a one-dimensional finite-state automaton when a string is the input to them, it is clear that $L \notin L(DFSA) \cup L(DOTA)$. The inclusion $L(FSA) \subseteq L(DFA)$ was proved in [10], the remaining inclusions hold by Proposition 10. □

6.2 Forgetting Automata and NP_{2d}-completeness

Proposition 11 $L(DFA) \subseteq P_{2d}$

Proof. Let $A = (Q, \Sigma, q_0, \delta, Q_F)$ be a two-dimensional deterministic forgetting automaton and $P \neq \Lambda$ be an input to it of size $m \times n$. We can easily show that if A computing over P halts, it performs maximally $c \cdot (m \cdot n)^2$ steps, where c is a constant independent on P. The head of A can be placed at $m \cdot n + 2 \cdot (m + n)$ different positions (fields corresponding to P and also columns of #'s neighboring to borders of P). The control unit can be in one of $|Q|$ different states. That implies that during $s = |Q| \cdot [m \cdot n + 2 \cdot (m + n)]$ steps A must erase at least one symbol, otherwise it repeats the same configurations and cycles. There are $m \cdot n$ symbols that can be erased, thus the number of steps A performs is bounded by $s \cdot m \cdot n$ which is $O(m^2 \cdot n^2)$. □

Proposition 12 *There is a FA recognizing an NP_{2d}-complete language.*

Proof. We will show that it is possible to construct FA A recognizing SAT problem.

First of all we describe how instances of the problem will be encoded. Let the input alphabet be $\Sigma = \{0, 1, \wedge, \vee, \neg, *\}$, $F = \bigwedge_{i=1}^{m} (\bigvee_{j=1}^{s_i} A_{ij})$ be a formula (each A_{ij} is a variable or a negated variable). Let $n = \sum_{j=1}^{m} s_i$, V be the set of all different variables that appear in F and let $k = \lfloor \log_2 |V| \rfloor$. We can encode each variable in V by a binary string over $\{0, 1\}$ of length k. The whole formula F is encoded by a picture P of size $(2^k + 1 + 3 \cdot n) \times (k + 1)$. P consists of two parts – the left part which is formed of the first 2^k columns and the right part formed of the last $3 \cdot n$ columns. Both parts are separated by one additional column located between them. This column is filled by $*$'s. In the left part, each i-th column stores a string of length $k + 1$ representing $i - 1$ written in binary. It means, the first column is 0^{k+1}, the second column is $0^k 1$, etc. The right part represents F written as

$$A_{1,1} \vee \ldots \vee A_{1,s_1} \wedge A_{2,1} \vee \ldots \vee A_{2,s_2} \wedge \ldots \wedge A_{m,1} \vee \ldots \vee A_{m,s_m} \quad (1)$$

It is a concatenation of n blocks, each of them consisting of three columns. The i-th block represents the i-th A_{pq} in (1), possibly preceded by the negation sign, and the operator that immediately follows the variable (except the last variable, which is not followed by any operator). Let the j-th variable in (1) be encoded by a binary string $w = b_1 \ldots b_k$ and ϕ be the operator following it. Then, the first column of the j-th block stores $\neg b_1 \ldots b_k$ if the variable is negated, otherwise $0 b_1 \ldots b_k$, the second column is filled by 0's and the third column stores $\phi 0^k$. An example of a formula encoded by a picture is shown in Figure 6.3.

				$*$	$\neg$		$\vee$			$\wedge$			$\vee$	$\neg$		
0	0	1	1	$*$	0			0			1			0		
0	1	0	1	$*$	0			1			0			1		

Figure 6.3: $(\neg A \vee B) \wedge (C \vee \neg B)$ encoded by a picture. All fields that are not labelled by any symbol store 0.

Let P' be a picture over Σ. It is possible to check by a $DFSA$ A_2 whether P' encodes a formula or not. It can be done checking the left and right part as follows.

- The left part is checked column by column starting by the leftmost one. First of all, A_2 verifies if the first column is completely filled by 0's. After that, A_2 repeatedly checks if the next column represents a binary value and if this value equals the value represented in the previous column incremented by one. To do it, A_2 moves the head in the previous column from the least significant bit to the most significant bit, performs "plus one" operation in states and verifies continually if the computed bits match values in the column that is being checked. The end of the left part is given by a column storing string of the form 01^k. This column has to be followed by the separating column formed of $*$'s.
- In the right part, A_2 checks if each block contains in particular fields symbols that are permitted by the used encoding.

Now, we can proceed with the algorithm for A. Let the input to A be P'. A computes as follows.

- A checks if P' encodes a formula. If not, A rejects P'.
- A goes trough all variable codes represented in the left part of P' and non-deterministically assigns true or false value to each of them. When a value is assigned to a particular code, A goes trough blocks in the right part of P' and guesses which of the blocks store the variable corresponding to the code. Whenever a block is chosen, A checks if the stored code really matches the code in the left part. If it does not, A rejects, otherwise it records the assigned boolean value in the block. The first field of the middle column is used to do it – this field is erased just in the case of true value.
- A checks that each block was chosen during some stage of the previous procedure. If so, it evaluates the formula, otherwise it rejects. This can be done easily in states while scanning the first row of the right part. A accepts if and only if the formula is evaluated to true.

To complete the description, it remains to give more details on the second part of the algorithm.

Whenever A finishes guessing of blocks that match the currently processed code in the left part, it moves the head to the column storing this code and erases all fields in the column except the first one. The code to be processed next is then the leftmost column not containing erased symbols thus it can be detected

				$*$	$\neg$		$\vee$		@	$\wedge$			$\vee$	$\neg$	@	
@	@	1	1	$*$	0	@	@	0	@	@	1		@	0	@	@
@	@	0	1	$*$	0	@	@	1	@	@	0		@	1	@	@

Figure 6.4: Computation over the picture encoding $(\neg A \vee B) \wedge (C \vee \neg B)$ in progress. The automaton has completed evaluation and guessing of occurrences of variables A and B (they were guessed correctly, A evaluates to false, while B to true). Variable C encoded by 10 is being processed now. Moreover, the third block has been guessed to contain an occurrence of it (at this stage, within the block, symbols are erased in the third column only).

easily. Moreover, whenever A verifies that a block in the right part matches the currently processed code and records a boolean value in the block, the block is always left with all fields in the second and third column, except the first row, being erased. On the other hand, a block that has not been guessed to store a variable matching the currently processed code has all the mentioned fields non-erased. This helps to distinguish between already evaluated and non-evaluated blocks.

Now, let us assume A has already processed some of the codes and it is about to start to process the next one. It non-deterministically chooses a boolean value for the correspondent variable and remembers it in states. It places the head in the first block and then, while moving it left, it guesses if the scanned block matches the current code (note that this guess is done only if the block has not been already matched to one of the previously processed codes and evaluated). If a block is chosen, A erases all symbols in its third column except the first one. After that, A repeatedly checks bit by bit of the code represented in the block if it matches the correspondent bit in the left part. It starts in the last row. In states, it records the least significant bit, moves the head left until it detects the leftmost non-erased field in the left part (when A encounters $*$, it is notified about entering the left part). It verifies if the bit recorded in states is equal to the scanned bit. After that, it moves the head one row up and performs the verification on the next pair of bits. Note that A is able to detect the block in the right part currently being processed, since it is the only block containing in the scanned row non-erased symbol in the middle column and @ in the third column. When the verification is done, A records the chosen boolean value in the first field of the second column of the block and erases the remaining fields of the column. Then it continues in moving the head right and guesses the other occurrences of the variable. See Figure 6.4 for a scheme illustrating a computation of A in progress.

It should be clear that the described algorithm is correct. If the encoded formula is satisfiable, A can guess a suitable evaluation of variables as well as occurrences of variables in the blocks. On the other hand, if A manages to assign boolean values to all blocks, this assignment is always consistent – whenever two blocks store the same variable code, the same boolean value has been assigned to them. □

In the proof, we encoded F by a picture of size $(2^k + 1 + 3 \cdot n) \times (k + 1)$. It means, we needed $O(|V| \cdot \log |V| + n \cdot \log |V|) = O(n \cdot \log |V|)$ fields to do it.

Let the language the automaton A recognizes be denoted by L_{SAT}. In the presented construction, we used the advantage of the two-dimensional topology. Binary codes of variables were represented in columns, thus a check if the i-th bits of two codes equal or not could have been simply done moving the head in a row. We would have difficulties if we try to design a one-dimensional forgetting automaton recognizing $\gamma(L_{SAT})$ (see section 4.1 for the definition of γ). To make an analogous construction for one-dimensional FA, it would require to use different encoding of formulas that will provide space large enough to perform the checks. Another possibility is to encode variables in unary which requires strings of length $O(n \cdot |V|)$. Such a construction has been done by F. Mraz for one-dimensional forgetting automata with the operation of deletion.

Note that we have not explicitly shown that L_{SAT} is NP_{2d}-complete, however, it is implied by Theorem 5, since $\gamma(L_{SAT})$ is surely an NP-complete language.

Since it is very unlikely that $NP = P$, based on the previous two propositions, we can state the following conjecture.

Conjecture 1 *$L(DFA)$ is a proper subset of $L(FA)$.*

Furthermore, since the language in Lemma 7 is not in $L(OTA)$, we know that $L(DFA)$ is not a subset of $L(OTA)$. On the other hand, by Proposition 11 and Proposition 9, $NP \neq P$ implies $L(OTA)$ is not subset of $L(DFA)$.

Conjecture 2 *$L(DFA)$ and $L(OTA)$ are incomparable.*

Chapter 7

Grammars with Productions in Context-free Form

7.1 Introduction into Two-dimensional Context-free Grammars

One of the motivations why the grammars we deal with in this chapter are studied comes from M.I. Schlesinger and V. Hlavac. They present the grammars in [22] as a tool for pattern recognition based on syntactic methods.

We will follow their informal description leading to the notion of two-dimensional context-free grammars and giving basic ideas about their usage in the mentioned area.

Let us consider the set of all pictures over the alphabet $\Sigma = \{b, w\}$, where b, resp. w represents a pixel of the black, resp. white color. Our task will be to define a sub-set of pictures that can be considered to contain exactly the letter H written in black on white (in [22], Schlesinger and Hlavac work with the Russian letter "sh"). We require the definition to be based on rules of the following forms:

1. A picture of size 1×1 is named b, resp. w if its the only field is of the black, resp. white color.
2. A picture is named s if it can be divided into two parts by a horizontal line so that the top part is named s_t and the bottom part is named s_b.
3. A picture is named s if it can be divided into two parts by a vertical line so that the left part is named s_l and the right part is named s_r.
4. A picture is named s if it is also named s'.

The presented rules are simple enough to be checked automatically by a recognizing device.

We would like to form a suitable set of rules determining which pictures are named H. The first rule is demonstrated by Figure 7.1. A picture is named H

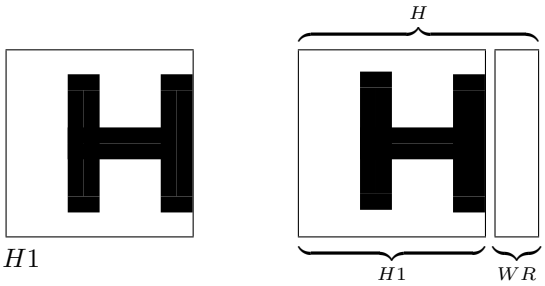


Figure 7.1: Generating the letter H – a usage of productions (1.1) and (1.2), i.e. the separation of a white border on the right.

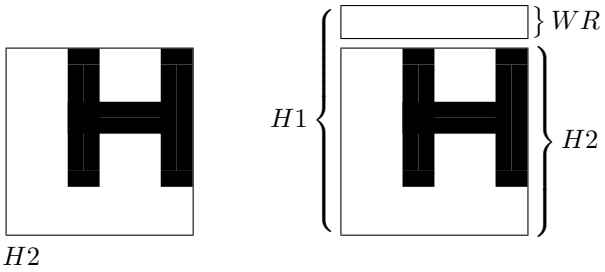


Figure 7.2: Generating the letter H – a usage of productions (2.1) and (2.2), i.e. the separation of a white border on the top.

if it can be divided by a vertical line into two pictures, where the right part is a white rectangle (a picture named WR) and the left part is a picture representing H such that the letter is "attached" to the right border of the picture (a picture named $H1$). Also, a picture is named H if it is named $H1$. This rule covers situations, where there is no white rectangle neighboring with the letter on the right side. We denote these two rules by assignments as follows:

$$H := H1 \qquad (1.1)$$
$$H := H1 \,|\, WR \qquad (1.2)$$

We can continue with pictures named $H1$ analogously – they can be divided by a horizontal line into two parts, where the top (if present) is a white rectangle and the bottom a picture named $H2$ storing the letter H attached to the top-right corner. After that we can cut a white rectangle on the left (we get $H3$) and finally on the bottom (we get $H4$). The corresponding rules, illustrated in Figures 7.2, 7.3 and 7.4, follow:

$$H1 := H2 \qquad (2.1) \qquad\qquad H2 := H3 \qquad (3.1)$$
$$H1 := \tfrac{WR}{H2} \qquad (2.2) \qquad H2 := WR \,|\, H3 \qquad (3.2)$$
$$H3 := H4 \qquad (4.1)$$
$$H3 := \tfrac{H4}{WR} \qquad (4.2)$$

A picture named $H4$ is decomposed into a vertical concatenation of pictures BR and T, where BR is a black rectangle. T is formed of V and BR, V is a horizontal concatenation of WR and $V1$ and finally, $V1$ is decomposed into BR and WR – see Figure 7.5. Rules follow:

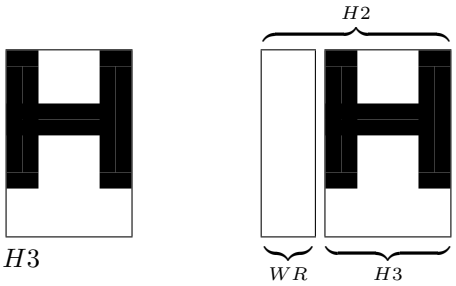


Figure 7.3: Generation of the letter H – a usage of productions (3.1) and (3.2), i.e. the separation of a white border on the left.

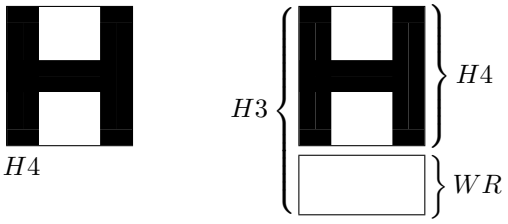


Figure 7.4: Generation of the letter H – a usage of productions (4.1) and (4.2), i.e. the separation of a white border on the bottom.

$$H4 := T\,|\,BR \quad (5.1) \qquad T := BR\,|\,V \quad (5.2)$$
$$V := \frac{WR}{V1} \quad (5.3) \qquad V1 := \frac{BR}{WR} \quad (5.4)$$

It remains to give rules for pictures named WR and BR respectively:

$$WR := \frac{WR}{WR} \quad (6.1) \qquad BR := \frac{BR}{BR} \quad (7.1)$$
$$WR := WR\,|\,WR \quad (6.2) \qquad BR := BR\,|\,BR \quad (7.2)$$
$$WR := w \quad (6.3) \qquad BR := b \quad (7.3)$$

See related Figures 7.6 and 7.7.

Figure 7.8 shows an example of a picture of the letter H, which can be named H using the presented rules, as well as an example of a picture which does not contain H and which cannot be named H applying the rules. However, the same Figure demonstrates that our definition is not fully ideal. We

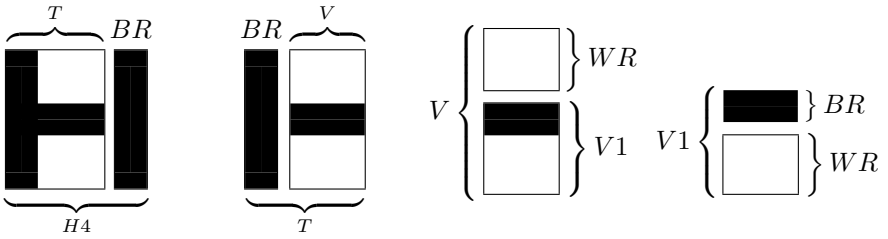


Figure 7.5: Generating the letter H – a usage of productions (5.1), (5.2), (5.3) and (5.4).

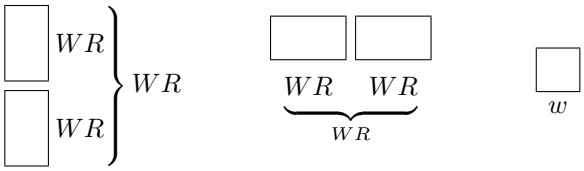


Figure 7.6: Generation of the letter H – a usage of productions (6.1), (6.2) and (6.3), i.e. a decomposition of a white rectangle.

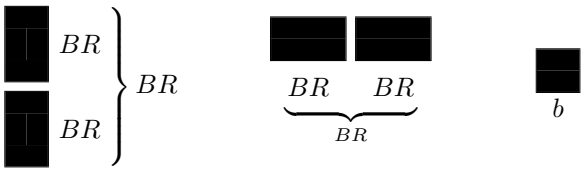


Figure 7.7: Generating the letter H – a usage of productions (7.1), (7.2) and (7.3), i.e. a decomposition of a black rectangle.

can see a picture named H which can be hardly considered to be H and, on the other hand, a picture containing H with some small fluctuations that cannot be named H using the rules. Our example is not suitable to be used for pattern recognition directly. Its purpose is mainly to show tools on which more sophisticated approaches can be based.

We have seen four types of context-free rules. These rules can be translated into productions as follows:

$$N \to \ A \quad B \qquad N \to \begin{matrix} A \\ B \end{matrix} \qquad N \to A \qquad N \to a$$

N, A and B are non-terminals, a is a terminal. There is a possibility to go beyond these simple right-hand sides containing one or two elements only – a general matrix can be considered there, productions are of the form $N \to [A_{ij}]_{p,q}$ then. The context-freeness is still preserved. This form can be considered as the generalization of the form of one-dimensional context-free productions.

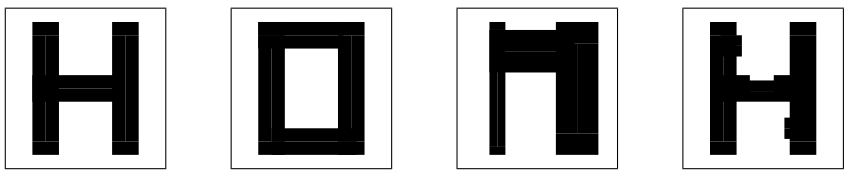


Figure 7.8: (a) a picture resembling the letter H that can be generated by the presented productions; (b) a picture not resembling H that cannot be generated; (c) a picture not resembling H that can be generated; (d) a picture resembling H that cannot be generated

Schlesinger and Hlavac do not study context-free grammars with general right-hand sides, but we can find another source, where these grammars appear – they were introduced by P. Jiricka [9] in his master thesis. Comparing to the example given above, a different approach to define pictures generated by a context-free grammar is used there. It is based on sentential forms and the relation "derives" rather than the presented recurrent approach. To complete the list, we should mention O. Matz [13] who also studies context-free grammars with single right-hand sides. His paper focuses on connections between the grammars and a sort of two-dimensional regular expressions.

Results in [22], [9] and [13] cover a few basic properties of the class generated by context-free grammars only. Schlesinger and Hlavac give an algorithm solving the membership problem in polynomial time (an analogy to the algorithm known from the one-dimensional case), P. Jiricka shows how a two-dimensional context-free language can be recognized by a FA. In the next sections we will give a more complex characterization of the class. It will include the following facts:

- Both approaches of how to define generated pictures (i.e. the recurrent one and that one based on sentential forms) lead to the same class.
- The generative power of context-free grammars having restricted right-hand sides of their productions (Schlesinger, Hlavac) is less than the generative power of context-free grammars without the restriction.

We will also see that there are many properties of the class that do not resemble the properties of the class of one-dimensional context-free languages. That is the reason why we do not use the term *context-free grammars*, but rather *grammars with productions in context-free form*. We will keep on using the second term strictly from now.

As for the other proposals of two-dimensional context-free grammars, resp. languages, we will show two of them in the end of this chapter (section 7.9). We will also make a basic comparison between these languages and $L(CFPG)$.

7.2 CFP Grammars, Derivation Trees

In this section we give a formal definition of the grammars with productions in context-free form and the languages generated by them. We also introduce derivations trees for the grammars.

Definition 11 A two-dimensional grammar with productions in context-free form *is a tuple* $(V_N, V_T, S_0, \mathcal{P})$*, where*

- V_N *is a finite set of non-terminals*
- V_T *is a finite set of terminals*
- $S_0 \in V_N$ *is the initial non-terminal*
- $\mathcal{P}$ *is a finite set of productions of the form* $N \to W$*, where* $N \in V_N$ *and* $W \in (V_N \cup V_T)^{**} \setminus \{\Lambda\}$*. In addition,* $\mathcal{P}$ *can contain* $S_0 \to \Lambda$*. In this case, no production in* $\mathcal{P}$ *contains* S_0 *as a part of its right-hand side.*

Definition 12 *Let $G = (V_N, V_T, S_0, \mathcal{P})$ be a two-dimensional grammar with productions in context-free form. We define a picture language $L(G, N)$ over V_T for every $N \in V_N$. The definition is given by the following recurrent rules:*

A) If $N \to W$ is a production in $\mathcal{P}$ and $W \in {V_T}^{**}$, then W is in $L(G, N)$.

B) Let $N \to [A_{ij}]_{m,n}$ be a production in $\mathcal{P}$, different to $S_0 \to \Lambda$, and P_{ij} $(i = 1, \ldots, n;\ j = 1, \ldots, m)$ be pictures such that

- *if A_{ij} is a terminal, then $P_{ij} = A_{ij}$*
- *if A_{ij} is a non-terminal, then $P_{ij} \in L(G, A_{ij})$*

Then, if $\bigoplus [P_{ij}]_{m,n}$ is defined, $\bigoplus [P_{ij}]_{m,n}$ is in $L(G, N)$.

The set $L(G, N)$ contains exactly all pictures that can be obtained by applying a finite sequence of rules A) and B). The language $L(G)$ generated by the grammar G is defined to be the language $L(G, S_0)$.

We abbreviate a two-dimensional grammar with productions in context-free form by $CFPG$ (or by CFP grammar). $L(CFPG)$ is the class of all languages generated by these grammars. Languages in $L(CFPG)$ are called CFP languages. If $P \in L(G, N)$, we say N *generates* P in G, or shortly, N generates P if it is evident from the context, which grammar G we refer to.

To illustrate the presented definition, we give a simple example of a $CFPG$ generating the set of all non-empty square pictures over a one-symbol alphabet.

Example 7 Let $G = (V_N, V_T, S_0, \mathcal{P})$ be a CFP grammar, where $V_T = \{a\}$, $V_N = \{V, H, S_0\}$ and $\mathcal{P}$ contains the following productions:

1) $H \to a$ 2) $H \to a\ H$ 3) $V \to a$ 4) $V \to \begin{matrix} a \\ V \end{matrix}$

5) $S_0 \to a$ 6) $S_0 \to \begin{matrix} a & H \\ V & S_0 \end{matrix}$

Productions 1), 2) are one-dimensional, thus it should be clear that $L(G, H)$ contains exactly all non-empty rows of a's – applying rule A) on production 1), we have $a \in L(G, H)$. Furthermore, if $a^k \in L(G, H)$, then rule B) applied on production 2) gives $a^{k+1} \in L(G, H)$. Similarly, $L(G, V)$ contains non-empty columns of a's.

Applying rule A) on production 5), we get that a is generated by G. Since $a \in L(G, S_0) \cap L(G, H) \cap L(G, V)$, rule B) applied on production 6) gives that the square 2×2 is also in $L(G, S_0)$. The row, resp. column of length 2 is generated by H, resp. V, thus rule B) can be applied again to produce the square 3×3, etc. By induction on the size, we can show that each non-empty square picture over $\{a\}$ can be generated and that there is no way to generate any non-square picture.

Definition 13 *Let $G = (V_N, V_T, S_0, \mathcal{P})$ be a $CFPG$. A derivation tree for G is every tree T satisfying:*

- *T has at least two nodes.*

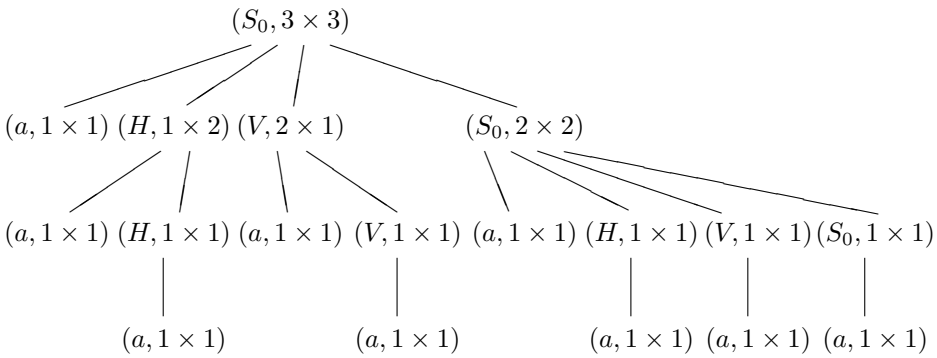


Figure 7.9: Example of a derivation tree T, where $p(T)$ is the square 3×3 over $\{a\}$. Nodes are represented by assigned labels, edges appear in the lexicographical order which is induced by their labels.

- *Each node v of T is labelled by a pair $(a, k \times l)$. If v is a leaf then $a \in V_T$ and $k = l = 1$ else $a \in V_N$ and k, l are positive integers.*
- *Edges are labelled by pairs (i, j), where $i, j \in \mathbb{N}^+$. Let $I(v)$ denote the set of labels of all edges connecting v with its children. Then, it holds that $I(v) = \{1, \ldots, m\} \times \{1, \ldots, n\}$ and $m \cdot n$ is the number of descendants of v.*
- *Let v be a node of T labelled by $(N, k \times l)$, where $I(v) = \{1, \ldots, m\} \times \{1, \ldots, n\}$. Let the edge labelled by (i, j) connect v and its child v_{ij} labelled by $(A_{ij}, k_i \times l_j)$. Then, $\sum_{i=1}^{m} k_i = k$, $\sum_{j=1}^{n} l_j = l$ and $N \to [A_{ij}]_{m,n}$ is a production in $\mathcal{P}$.*

If $S_0 \to \Lambda \in \mathcal{P}$ then the tree T_Λ with two nodes – the root labelled by $(S_0, 0 \times 0)$ and the leaf labelled by $(\Lambda, 0 \times 0)$ – is a derivation tree for G too (the only edge in T_Λ is labelled by $(1, 1)$).

Let T be a derivation tree for a CFP grammar $G = (V_N, V_T, S, \mathcal{P})$ such that $T \neq T_\Lambda$, V be the set of its nodes. We assign a picture to each node of T by defining a function $p : V \to {V_T}^{**}$ as follows: if $v \in V$ is a leaf labelled by $(a, 1 \times 1)$ then $p(v) = a$ else $p(v) = \bigoplus [P_{ij}]_{m,n}$, where $I(v) = \{1, \ldots, m\} \times \{1, \ldots, n\}$, $P_{ij} = p(v_{ij})$, v_{ij} is the child of v connected by the edge labelled by (i, j). We also define $p(T)$ to be $p(r)$, where r is the root of T, moreover, $p(T_\Lambda) = \Lambda$.

We can make a simple observation: if $v \in V$ is labelled by $(N, k \times l)$ then $\mathrm{rows}(p(v)) = k$ and $\mathrm{cols}(p(v)) = l$.

Figure 7.9 shows an example of a derivation tree which corresponds to generating the square 3×3 in Example 7.

Lemma 8 *Let $G = (V_N, V_T, S, \mathcal{P})$ be a CFP grammar and $N \in V_N$.*

1. *Let T be a derivation tree for G having its root labelled by $(N, k \times l)$. Then $p(T) \in L(G, N)$.*
2. *Let P be a picture in $L(G, N)$. There is a derivation tree for G with root labelled by $(N, k \times l)$ such that $\mathrm{rows}(P) = k$, $\mathrm{cols}(P) = l$ and $p(T) = P$.*

Proof. The lemma follows directly from the previous definitions. □

7.3 $L(CFPG)$ in Hierarchy of Classes

Example 8 Let us define a picture language L over $\Sigma = \{a, b\}$ as follows:

$$L = \{P \mid P \in \{a,b\}^{**} \wedge \exists i,j \in \mathbb{N} : \ 1 < i < \mathrm{rows}(P) \ \wedge \ 1 < j < \mathrm{cols}(P) \ \wedge \\ \forall x \in \{1,\ldots,\mathrm{rows}(P)\}, y \in \{1,\ldots,\mathrm{cols}(P)\} : \ P(x,y) = a \Leftrightarrow x \neq i \ \wedge \ y \neq j\}$$

Informally, a picture P over Σ is in L iff P contains exactly one row and one column completely filled by b's and this row, resp. column is not the first neither the last row, resp. column of the picture. An example follows:

$$\begin{array}{cccccc} a & b & a & a & a & a \\ a & b & a & a & a & a \\ b & b & b & b & b & b \\ a & b & a & a & a & a \end{array}$$

L belongs to $L(CFPG)$. It is generated by the CFP grammar $G = (V_N, \Sigma, S, \mathcal{P})$, where $V_N = \{S, A, V, H, M\}$ and the set $\mathcal{P}$ consists of the following productions:

$$S \to \begin{array}{ccc} A & V & A \\ H & b & H \\ A & V & A \end{array} \qquad A \to M \qquad A \to \ A \ \ M \qquad M \to a \qquad M \to \begin{array}{c} a \\ M \end{array}$$

$$V \to b \qquad V \to \begin{array}{c} b \\ V \end{array} \qquad H \to b \qquad H \to \ b \ \ H$$

The non-terminal M generates one-column pictures of a's, A generates the language $\{a\}^{**} \setminus \{\Lambda\}$, V generates one-column pictures of b's and finally, H generates one-row pictures of b's.

Definition 14 *Let $G = (V_N, V_T, S, \mathcal{P})$ be a CFP grammar. We say G is $CFPG2$ iff each production in $\mathcal{P}$ has one of the forms: $N \to a$, $N \to \Lambda$, $N \to A$, $N \to [A_{ij}]_{1,2}$ or $N \to [A_{ij}]_{2,1}$, where a is a terminal, A and each A_{ij} non-terminals.*

$CFPG2$ grammars are the grammars studied by Schlesinger and Hlavac [22]. Let $L(CFPG2)$ denote the class of languages generated by them. We will prove that $L(CFPG2)$ is strictly included in $L(CFPG)$.

In proofs that follow, we will often use words *a picture P is generated in the first step by a production* Π. Assuming we refer to a grammar G, the interpretation should be as follows: there is a derivation tree T for G such that $p(T) = P$ and Π is the production connected to the root of T.

Theorem 12 *$L(CFPG2)$ is a proper subset of $L(CFPG)$.*

Proof. By contradiction. Let $G = (V_N, V_T, S, \mathcal{P})$ be a $CFPG2$ generating the language L in Example 8. Let n be an integer greater than 2 (chosen with respect to the requirements that will be given), L_1 be the subset of L containing exactly all square pictures of size n in L. We can choose n to be sufficiently large so that no picture in L_1 equals the right-hand side of an arbitrary production in $\mathcal{P}$. L_1 consists of $(n-2)^2$ pictures. At least $\lceil \frac{(n-2)^2}{|\mathcal{P}|} \rceil$ of these pictures are generated in the first step by the same production. Without loss of generality, let the production be $S \rightarrow A\,B$. If n is large enough, then there are two pictures with different indexes of the row of b's (maximally $n-2$ pictures in L_1 can have the same index). Let us denote these pictures by O and $\overline{O}$. It holds $O = O_1 \oplus O_2$, $\overline{O} = \overline{O}_1 \oplus \overline{O}_2$, where $O_1, \overline{O}_1 \in L(G, A)$ and $O_2, \overline{O}_2 \in L(G, B)$ which implies $O = O_1 \oplus \overline{O}_2 \in L(G)$. It is a contradiction, since O contains b in the first and and also in the last column, but these b's are not in the same row. □

We have found later that O.Matz [13] already showed there is a language in $L(DFSA)$ that cannot be generated by any $CFPG2$. His example of the language is similar to that one presented in the proof.

The technique of proof based on categorizing pictures by a production used to generate them in the first step will be used several times in the following text, so, the purpose of the proof was also to introduce the technique on a simple language. The other usages will be more complicated.

Example 9 Let us define a language L over the alphabet $\Sigma = \{0, 1, x\}$, where a picture $P \in \Sigma^{**}$ is in L if and only if:

1. P is a square picture of an odd size
2. $P(i,j) = x \Leftrightarrow i, j$ are odd indexes
3. if $P(i,j) = 1$ then the i-th row or the j-th column (at least one of them) consists of 1's

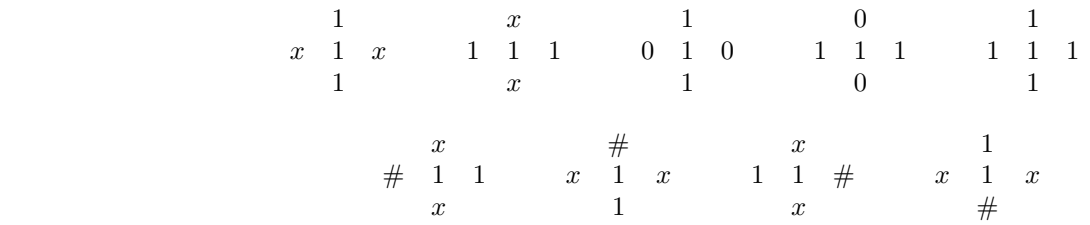
Here is an example of a picture in L:

$$\begin{array}{ccccccc} x & 1 & x & 1 & x & 0 & x \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ x & 1 & x & 1 & x & 0 & x \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ x & 1 & x & 1 & x & 0 & x \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ x & 1 & x & 1 & x & 0 & x \end{array}$$

Lemma 9 *L is recognizable by a $DFSA$.*

Proof. A $DFSA$ automaton A recognizing L can be constructed as follows. A checks if the input picture is a square picture of an odd size. It can be done by counting mod 2 when moving the head diagonally. After this verification, A scans row by row and checks if the symbol x is contained exactly in all fields having both indexes of odd values and in all cases when a field contains the

symbol 1, whether the four neighboring fields form one of the following allowed configurations:



This local check ensures that the third property is fulfilled. □

Lemma 10 *$L(DFSA)$ is not a subset of $L(CFPG)$.*

Proof. Let $G = (V_N, V_T, S, \mathcal{P})$ be a $CFPG$ such that $L(G) = L$, where L is the language from Example 9. Without loss of generality, $\mathcal{P}$ does not contain any production of the form $A \to B$, where A, B are non-terminals. We take an odd integer $n = 2k + 1$ which conforms requirements listed in the next paragraphs.

Let L_1 be the set of all pictures in L of size n. We can consider n to be chosen sufficiently large so that no picture in L_1 equals the right-hand side of an arbitrary production in $\mathcal{P}$. We have $|L_1| = 2^k \cdot 2^k = 2^{n-1}$ (there are k columns and k rows, where, for each of them, we can choose if the whole row, resp. column consists of 1's or not). L_1 contains at least $\lceil \frac{2^{n-1}}{|P|} \rceil$ pictures that can be generated in the first step by the same production. Let the production be $S \to [A_{ij}]_{p,q}$ and let the mentioned set of pictures be L_2. Without loss of generality, we assume $p \geq q$ and moreover, $p \geq 2$ (otherwise the production is of the form $A \to B$).

Now, our goal is to show there are two pictures $U, V \in L_2$ that can be written as $U = \bigoplus [U_{ij}]_{p,q}$, $V = \bigoplus [V_{ij}]_{p,q}$, where all U_{ij}, V_{ij} are in $L(G, A_{ij})$ and $\mathrm{rows}(U_{ij}) = \mathrm{rows}(V_{ij})$, $\mathrm{cols}(U_{ij}) = \mathrm{cols}(V_{ij})$ (we denote this property by (1)) and furthermore, that the first row of U does not equal the first row of V (let this be property (2)). The number of all different sequences

$$\mathrm{cols}(U_{1,1}), \mathrm{cols}(U_{1,2}), \ldots, \mathrm{cols}(U_{1,q}), \mathrm{rows}(U_{1,1}), \mathrm{rows}(U_{2,1}), \ldots, \mathrm{rows}(U_{p,1})$$

is bounded by n^{p+q}. It implies there is a subset $L_3 \subseteq L_2$ such that

$$|L_3| \geq \frac{2^{n-1}}{|\mathcal{P}| \cdot n^{p+q}}$$

and each pair of pictures in L_3 fulfills property (1). Let L' be a subset of L_2, where arbitrary two different pictures do not satisfy property (2). We have $|L'| \leq 2^k = 2^{\frac{n-1}{2}}$ (columns of even indexes that are completely filled by 1 are determined by the fixed first row, thus we can choose among k rows of even index which of them to completely fill by 1). We can assume n is sufficiently large so that

$$|L_3| > 2^{\frac{n-1}{2}}$$

It implies the pair U, V we are looking for can be found in L_3. If we replace sub-pictures $U_{1,1}, \ldots, U_{1q}$ in U by sub-pictures $V_{1,1}, \ldots, V_{1q}$ (U_{1i} being replaced

by V_{1i}), we get a picture P which belongs to L again. It is a contradiction, since P does not have all properties of pictures in L. □

Since one-dimensional context-free grammars are a special case of CFP grammars, all context-free languages are included in $L(CFPG)$. It implies that $L(CFPG)$ is not a subset of $L(DFSA)$, neither of $L(OTA)$. We summarize this observation and the proved theorem:

Theorem 13 *$L(CFPG)$ is incomparable to $L(DFSA)$, $L(FSA)$, $L(DOTA)$ and $L(OTA)$.*

Proof. Follows from Lemma 10 and the observation. □

P. Jancar, F. Mraz and M. Platek have shown in [8] that the class of one-dimensional context-free languages can be characterized by one-dimensional forgetting automata. We extend this idea into two dimensions and study the relationship between FA's and CFP grammars. Our result is as follows: for a given CFP grammar G, it is possible to construct a FA recognizing $L(G)$. On the other hand, since $L(FSA) \subseteq L(FA)$, it is clear that FA's are stronger than GFP grammars and the model cannot be used as a recognizing device characterizing $L(CFPG)$.

The construction in the following proof is based on the construction that has been already presented by P. Jiricka in [9]. Comparing to this author, our version includes some improvements and simplifications. We also investigate some "edge" cases that are missing in [9].

Theorem 14 $L(CFPG) \subseteq L(FA)$

Proof. Let us consider a CFP grammar $G = (V_N, V_T, S_0, \mathcal{P})$. We describe how to construct a two-dimensional forgetting automaton A accepting $L(G)$. Without loss of generality, we assume $\mathcal{P}$ does not contain any production of the form $A \to B$, where $A, B \in V_N$. Let the working alphabet of A be $\Sigma = V_T \cup \{\#, @\}$ and O be an input to A. The idea of the computation is to try to build a derivation tree T in G having the root labelled by $(S_0, \mathrm{rows}(O) \times \mathrm{cols}(O))$ and $p(T) = O$.

Let M be the portion of the tape storing O in the initial configuration. By a *region* we mean each block $B \subseteq M$ labelled by an element in $V_T \cup V_N$. We distinguish two types of regions:

- t-region is a region consisting of one field and having assigned an element in V_T as the label
- N-region is a region labelled by an element in V_N (there is no restriction on its size)

We say that a region is *represented* if there is information stored in the tape determining the position, size and label of the region (details on how this representation is realized will be given later). The computation of A consists of several cycles. After finishing a cycle, let $\mathcal{S}$ be the set containing exactly all currently represented regions. It always holds that each two regions in $\mathcal{S}$ are disjunct and the union of all the regions equals M (we treat regions as blocks, when taking them as operands of operations like union and intersection, their

labels are not relevant at this moment). Moreover, if R is an N-region in $\mathcal{S}$ labelled by X, then $R \in L(G, X)$. At the beginning (before the first cycle is started), M is split into $\text{rows}(O) \cdot \text{cols}(O)$ represented t-regions, each region at position (i, j) being labelled by $O(i, j)$. A cycle can be roughly described as follows: A non-deterministically chooses a block B that is the union of some represented regions, where the number of regions is limited by a global constant. Furthermore, it chooses a non-terminal N and checks if N can generate B applying a sequence of productions on the selected represented regions. After finishing a cycle, if $\mathcal{S}$ contains exactly one region labelled by S_0, it is clear that O can be generated by G.

To store information, A uses the technique presented in Example 6. Let n be a suitable constant determining the decomposition into blocks. We will specify it later, when we collect all requirements on the storage capacity of blocks. We will assume $\text{rows}(O), \text{cols}(O) \geq n$. The rest of inputs will be discussed as a special case. To distinguish between referring a block of the decomposition and a block of M in general, let us denote the set of blocks related to the decomposition by $\mathcal{B}$.

We proceed by a description of how to represent regions during the computation. First of all, we will require that A does not represent any N-region that is a subset of a block in $\mathcal{B}$ – we denote this requirement by (1). It should be interpreted as follows: if $O \in L(G)$, then A can derive this fact keeping the requirement fulfilled upon finishing each cycle – we will show later how this can be done. The requirement ensures A need not to represent a large number of regions in a block. Since there can be $\Omega(n^2)$ regions of size 2×2, it would be problematic to fit correspondent information into the block. Instead of that, it is better to derive such a small region within a cycle when needed and combine it immediately with another regions to get a larger region.

In the following text, we work with N-regions fulfilling the requirement. Let us consider a block $B \in \mathcal{B}$ of size $k \times l$ and an N-region R (we remind that $n \leq k, l < 2n$). Let us denote the four fields neighboring with the corners of B as the *C-fields* of B (see Figure 7.10). We say that B is a *border* block of R iff $R \cap B \neq \emptyset$ and R does not contain all C-fields of B. A represents a region in its all border blocks.

We consider the bits available to store information in B to be ordered in some way and divided into groups. One group represents one intersection between B and some region such that B is a border of the region. Each group has an *usage flag* consisting of two bits determining if the group is *not used* (information has not been stored in the group yet), *used* (the group is currently used and represents an intersection) or *deleted* (stored information is not relevant anymore). The first state is indicated by two non-erased symbols, the second one by one symbol erased and finally the third one by two erased symbols. It allows to change the flag from 'not used' to 'used' and from 'used' to 'deleted'. The remaining bits of groups are reserved to represent coordinates in B (one coordinate is a value in $\{1, \ldots, 2n-1\}$, thus, stored in binary, it requires at most $2 \cdot \lfloor \log_2(2n) \rfloor$ bits), non-terminals (they can be represented in unary, which requires $|V_N|$ bits per one item) and some additional flags that will be listed in next paragraphs.

We say that an intersection between R and B is of the *first* type if R contains one or two C-fields of B and of the *second* type if R does not contain any C-field. It is obvious that if R has an intersection of the first, resp. second type

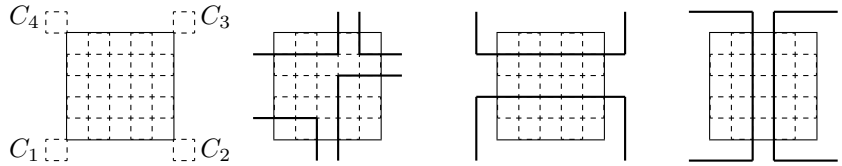


Figure 7.10: A block and its C-cells; eight different types of intersection between the block and an N_1-region.

with a border block then it has intersections of the same type with all its border blocks. It allows us to denote by N_1-region, resp. N_2-region each region having intersections of the first, resp. second type.

Let us inspect all possible configurations of an intersection between B and some N_1-region R. There are 8 different possibilities with respect to which C-fields of B are included in R – see Figure 7.10.

It means one intersection can be represented using 3 bits determining the type and one or two coordinates determining the exact position of the intersected part.

Next, let us solve the question how many intersections with N_1-regions A needs to represent during the computation in B. First of all, we will consider a situation, when B already represents an intersection with some N_1-region R_1 and a cycle is to be performed. If R_2 is a new N_1-region to be represented (a product of the cycle) such that $R_1 \subseteq R_2$ and R_2 has exactly the same intersection with B as R_1, then, to spare space in B, it is possible to reuse the old representation (A needs only to record a change of the label – we will describe in the next paragraph how this can be done). So, we have to estimate how many different intersections (with respect to type and coordinates) with represented N_1-regions can appear in B. To do it, we observe that after performing a cycle, B can be a border block of at most 4 different represented N_1-regions. The border (coordinates in B) of one N_1-region can be changed maximally $k+l-2$ times (before it completely leaves B), because every change increases, resp. decreases at least one of the coordinates. It means it is sufficient if B can represent $4 \cdot (k+l-2) \leq 4 \cdot (4n) \leq 16 \cdot n$ intersections.

While the border of an N_1-region R is represented in each border block, the label corresponding to R will be recorded in one of the border blocks only. We assume, each group of bits representing an intersection of the first type contains space to store one non-terminal together with a usage flag, which can be in one of the states 'not used', 'used' or 'deleted' again. If R is represented, exactly one of the usage flags is in the 'used' state. Then, the related non-terminal is stored in the correspondent block.

To show that this method is correct, we need to realize that whenever a new N_1-region is produced by a cycle and needs to be represented, then its border has at least one intersection of the first type, which is represented first time (i.e. it has not been reused from a region represented before, see Figure 7.11). So, an empty slot for the related non-terminal can be always found. Note that if A knows coordinates of R in B it can determine, which group of bits represents R in neighboring blocks, thus it is able to traverse trough all border blocks of R.

We consider N_2-regions to be vertical or horizontal (see Figure 7.10). Let the width of a vertical, resp. horizontal N_2-region be its number of columns,

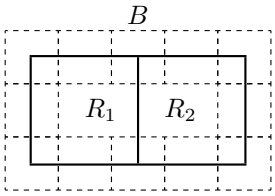


Figure 7.11: If the union of regions R_1, R_2 has to be represented, then the border block denoted by B contains a new intersection, thus a new group of bits in B will be used to represent it. Blocks in $\mathcal{B}$ are depicted by dashed lines.

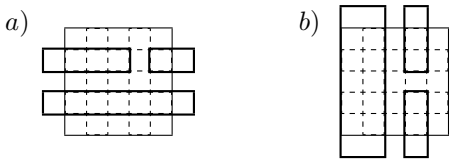


Figure 7.12: *a*) Three types of intersection between a horizontal N_2-region and a block. The intersection can contain the right end of the region, the left end or neither of them. *b*) Three types of intersection between a vertical N_2-region and a block.

resp. rows. We add the second requirement, denoted (2), on the representation of regions: If R is a represented horizontal, resp. vertical N_2-region then there was not any different represented horizontal, resp. vertical N_2-region of the same width having the same border blocks.

Without loss of generality, let R be a horizontal N_2-region. There are three types of intersection between R and B, thus the kind of an intersection can be represented using tree bits, where the first one determines if the region is vertical or horizontal and the second and third bit indicate one of the three possible types. Two coordinates are used to record positions of the first and last row of R. These coordinates are reused similarly as in case of N_1-regions. Also the representation of labels is done using the technique from the previous case. It is possible, since requirement (2) ensures that whenever a new N_2-region is to be represented, then it always has at least one new intersection comparing to already represented intersections, thus there can be found a group of bits that has not been used to store a non-terminal yet. The last information required to be recorded consists of coordinates of the left end of R in the leftmost border block and the right end in the rightmost border block. To spare space, these coordinates are handled together with the label. It means, the "usage flag and non-terminal" part of each group of bits representing an N_2-region is extended by space for two coordinates.

Now, we estimate how many different intersections with N_2-regions A needs to represent in a block B. We are interested in intersections that differ in type or in coordinates of the first and last row (horizontal N_2-regions), resp. columns (vertical N_2-region). Other coordinates are not relevant at this point. Let us consider horizontal intersections of some fixed kind and let $\mathcal{R}$ be the set of all different intersections of this kind that have to be represented in B. If I_1, I_2 are two different elements in $\mathcal{R}$, then one of these intersections covers the other

or I_1, I_2 are disjunct. We say I_1 directly covers I_2 if I_1 covers I_2 and there is not any $I_3 \in \mathcal{R}$ ($I_3 \neq I_1, I_3 \neq I_2$) such that I_3 covers I_2 and I_1 covers I_3. We can construct trees corresponding to the relation "directly covers". Leaves are intersections that do not cover any other intersections, children of a node are all directly covered intersections. There are at most $2n$ leaves which implies the trees have at most $4 \cdot n$ nodes, thus $|\mathcal{R}| \leq 4 \cdot n$. If we consider both types of N_2-regions and three kinds of intersections per a type, we can conclude that $6{\cdot}4{\cdot}n = 24{\cdot}n$ different intersections are needed to be represented in B maximally.

We complete and summarize what information should be stored in a block during the computation.

- One bit determines if B is a subset of some derived N-region or not – this information is changed during the computation once maximally. According to this bit, A determines if a field of a block that is not a border block of any N-region is a t-region or not.
- $16 \cdot n$ groups of bits are reserved to represent intersections of the first type. Each group consists of a usage flag, 3 bits determining the type of intersection and a subgroup storing one usage flag and label.
- $24 \cdot n$ groups of bits are reserved to represent intersections of the second type. Comparing to the previous type, the subgroup of these groups is extended by two coordinates.

It means we need $O(n \cdot \log(n))$ bits per a block, while a block can store $\Omega(n^2)$ bits, thus a suitable constant n can be found.

We can describe cycles in more details now. Let d be the maximal number of elements of the right-hand side of a production among productions in $\mathcal{P}$ and let $c = d \cdot 16 \cdot n^2$. In a cycle, A non-deterministically chooses a non-represented region R which is the union of some set of regions $\mathcal{R} = R_1, \ldots, R_s$ that are all represented, and a sequence of productions $P_1, \ldots, P_t$, where $s, t \leq c$. A chooses R as follows. It starts with its head placed in the top-left corner of O. It scans row by row from left to right, proceeding from top to bottom and non-deterministically chooses the top-left corner of R (it has to be the top-left corner of some already represented region). Once the corner is chosen, T moves its head to the right and chooses the top-right corner. While moving, when T detects a region R_i first time, it postpones the movement right and scans borders of R_i. It remembers in states what are neighboring regions of R_i including their ordering and also the label of R_i. After finishing the scan, it returns the head back to the position, where the movement right was interrupted, and continues. When the top-right corner is "claimed", A scans next rows of R until it chooses the last one. Every time T enters a new represented region (not visited yet), it detects its neighbors using the method we have already mentioned. Thanks to the recorded neighboring relation among R_i's, A is able to move its head from one region to any other desired "mapped" region.

A continues by deriving new regions. Note that all derivations are performed in states of A only. The first region is derived by P_1. A chooses $S_1 = \bigoplus [S_{ij}]_{s_1,t_1}$, where each S_{ij} is one of R_i's and checks if S_1 can be derived. Let us consider all S_{ij}'s to be deleted from $\mathcal{R}$ and S_1 to be added. A performs the second step on the modified set $\mathcal{R}$ using P_2, etc. In the last step, A derives R. After that, A records changes in the tape – representations of regions in $\mathcal{R}$ are deleted, while a representation of the new region R is created.

If the region corresponding to O labelled by S_0 is derived then T has been constructed and $O \in L$. On the other hand, let T be a derivation tree for G having its root labelled by $(S_0, \text{rows}(O) \times \text{cols}(O))$ and $p(T) = O$. A can verify whether $O \in L(G)$ despite the requirements (1), (2) as follows. The selection of regions in cycles is driven by the structure of T. Let us consider the nodes of T to be ordered in a sequence $\mathcal{V}$, where each parent is of a greater index than its any descendant (e.g. the post-order ordering). The sequence of regions to be represented determined by $\mathcal{V}$ as follows. The nodes are searched from the first one to the last one and a region R corresponding to a node v is chosen to be derived and represented iff

- R is not a subset of some border block (requirement (1))
- if R is an N_2-region, then, there is not any region R' corresponding to a node v' which is of a less index than v, such that $R' \subseteq R$, R and R' have the same border blocks and the same width (requirement (2))

For a node $v \in \mathcal{V}$, let $r(v)$ be the region corresponding to v. Let $R = r(v)$ be chosen to be produced in a cycle. We denote by $T(v)$ the subtree of T which is induced by v taken as the root and consists of all nodes located in paths from v to leaves. Next, we denote by $T(R)$ the only subgraph of $T(v)$ determined as follows:

- $T(R)$ is a tree, v is its root, a node v' in $T(v)$ is a leaf in $T(R)$ iff the region corresponding to v' is currently represented (meaning at the moment the cycle is started)
- each node and edge in $T(R)$ which is a part of the path from v to a leaf of $T(v)$ is in $T(v)$

Since each leaf corresponds to a region and each non-leaf node to a usage of one production in a cycle, our goal is to show that the number of leaves, resp. inner nodes of $T(R)$ is always bounded by c.

v has at most d descendants. If v' is one of them, then $r(v')$ is either a represented region, a region, which is a subset of some border block or non-represented N_2-region. If the second case occurs, then $r(v')$ contains at most $4n^2$ fields and thus $T(R)(v')$ has at most $4n^2$ leaves. In the third case, there must be a leaf v_1, a descendant of v' (possibly non-direct), such that $r(v_1)$ is a represented N_2-region of the same width and border blocks as $r(v')$ and $r(v_1) \subseteq r(v')$. It means the sub-tree $T(R)(v')$ has at most $2 \cdot (2n) \cdot (2n-1) + 1 \leq 8 \cdot n^2$ leaves, see Figure 7.13.

Summing over contributions of children of v, $T(R)$ can have $d \cdot 8 \cdot n^2$ leaves maximally implying the number of inner nodes is not greater than $16 \cdot d \cdot n^2 \leq c$, since, except those nodes that are parents of a leaf, each other inner node has at least two descendants. The case when a node has one descendant must correspond to productions of the form $A \to a$, where $a \in V_T$ (there are no productions $A \to B$, $B \in V_N$).

It remains to discuss the special case when either the width or height of O is less than n. If $\text{cols}(O)$ and $\text{rows}(O)$ are both less than n, A scans all symbols and computes in states whether O should be accepted. Without loss of generality, let $\text{cols}(O) = m < n$ and $\text{rows}(O) \geq n$. In this case, A needs to represent horizontal N_2-regions in a block only. In addition, $4 \cdot m$ different intersections

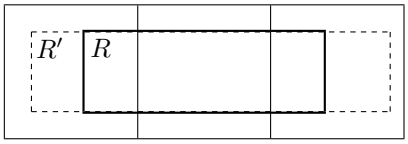


Figure 7.13: A horizontal region R intersects three blocks. R' is a region of the same type derived within a cycle. Comparing to R, its left, resp. right end is prolonged by at most $k \cdot (l-1) \leq (2n) \cdot (2n-1)$ fields on the left, resp. right.

have to be represented in a block maximally – this can be estimated similarly as in the case we have already inspected. $O(m \cdot \log(n))$ bits are required per a block, while $\Omega(m \cdot n)$ bits can be stored there, thus a suitable constant n exists again. □

Based on the previous theorem and $L(DFSA) \subseteq L(FA)$, it holds:

Corollary 1 *$L(CFGP)$ is a proper subset of $L(FA)$.*

M. Schlesinger and V. Hlavac proved that any language in $L(CFPG2)$ can be recognized in time $t(m,n) = O\left((m+n) \cdot m^2 \cdot n^2\right)$ by the RAM computational model. Their algorithm is the generalization of the well known algorithm for one-dimensional context-free languages which is of time complexity $t(n) = O(n^3)$ (and can be performed by a multi-tape Turing machine, as it can be found in [4]). We can make a simple generalization for $CFPG$, however, since there is no analogy to the Chomsky normal form (it will be proved in the following section), time complexity of the generalized algorithm is a polynomial which of degree depends on sizes of productions.

Theorem 15 $L(CFPG) \subseteq P_{2d}$

Proof. Let L be a language in $L(CFPG)$ and $G = (V_T, V_N, S, \mathcal{P})$ a grammar such that $L = L(G)$. Let p, resp. q be the maximal number of rows, resp. columns among the right-hand sides of productions in $\mathcal{P}$.

Let P be an input picture over V_T of size $m \times n$ and, for $x_1, x_2 \in \{1, \ldots, m\}$ and $y_1, y_2 \in \{1, \ldots, n\}$, where $x_1 \leq x_2$ and $y_1 \leq y_2$, let $P(x_1, x_2, y_1, y_2)$ denote the set containing exactly each field of P at coordinate (x, y), where $x_1 \leq x \leq x_2$ and $y_1 \leq y \leq y_2$. Such a set represents a rectangular area (a region) in P. We consider the areas to be ordered into a sequence $\mathcal{R}$, where, whenever R_1, R_2 are different areas and $R_1 \subseteq R_2$, then R_1 is of a lower index in $\mathcal{R}$ than R_2.

The algorithm goes through all areas in the order given by $\mathcal{R}$. For an area $R = P(x_1, x_2, y_1, y_2)$, it computes all non-terminals that generate the sub-picture of P represented by R. For every production $\Pi = N \to [A_{ij}]_{r,s}$ in $\mathcal{P}$, it inspects all decompositions of R into $r \times s$ sub-areas and checks whether the area at coordinate (i, j) can be generated by $A_{i,j}$ (when $A_{i,j} \in V_N$) or it consists of one field containing $A_{i,j}$ (when $A_{i,j} \in V_T$). The resulting set of non-terminals is recorded into a four-dimensional array, at index (x_1, x_2, y_1, y_2), so that the values can be used to compute the sets for the following areas.

We have, $|\mathcal{R}| \leq m^2 \cdot n^2$. Moreover, the number of decompositions of R is bounded by $m^{p-1} \cdot n^{q-1}$. It implies, for fixed G, the algorithm is of time

a	a	a	b	b	a	a	a	a
a	a	a	b	b	a	a	a	a
b	b	b	a	a	b	b	b	b
b	b	b	a	a	b	b	b	b
a	a	a	b	b	a	a	a	a

Figure 7.14: Example of a picture in $L(3)$.

complexity $O(m^{p+1} \cdot n^{q+1})$.

□

7.4 Restrictions on Size of Productions' Right-hand Sides

In Section 7.3 we have shown that if we restrict right-hand sides of productions to be of sizes 2×1, 1×2 and 1×1 only, then it is not possible to generate all languages in $L(CFPG)$. We can go further in these considerations and ask how the generative power of CFP grammars changes when sizes of right-hand sides are limited by a constant.

Definition 15 A CFP grammar with the restriction on productions' right-hand sides of the order k *($k \in \mathbb{N}^+$) is every $CFPG$ $G = (V_T, V_N, S, \mathcal{P})$, where each production $N \to [A_{ij}]_{m,n}$ in $\mathcal{P}$ fulfills: $m \leq k$ and $n \leq k$.*

We denote the class of CFP grammars with the restriction on productions' right-hand sides of the order k by $CFPG(k)$. $L(CFPG(k))$ is the class of languages generated by $CFPG(k)$ grammars.

Example 10 For any positive integer c, let $L_{ch}(c)$ be the language over $\Sigma = \{a, b\}$ consisting exactly of all pictures that can be written as $\bigoplus [P_{ij}]_{c,c}$, where

$$P_{ij} = \begin{cases} \text{a non-empty picture over } \{a\} \text{ iff } i+j \text{ is even} \\ \text{a non-empty picture over } \{b\} \text{ iff } i+j \text{ is odd} \end{cases}$$

Informally, each picture in $L_{ch}(c)$ can be interpreted as an irregular chessboard of size $c \times c$, formed of rectangles of a's and b's. An illustrative example is shown in Figure 7.14.

Theorem 16 *For each $k \in \mathbb{N}^+$, $L(CFPG(k))$ is a proper subset of $L(CFPG(k+1))$.*

Proof. The inclusion $L(CFPG(k)) \subseteq L(CFPG(k+1))$ is obvious, since each $CFPG(k)$ grammar is $CFPG(k+1)$ as well. To prove $L(CFPG(k)) \neq L(CFPG(k+1))$ we use the languages defined in Example 10 – we will show:

1. $L_{ch}(k+1) \in L(CFPG(k+1))$
2. $L_{ch}(k+1) \notin L(CFPG(k))$

Let us consider the grammar $G_1 = (\{a, b\}, \{A, B, S_1\}, S_1, \mathcal{P}_1)$, where $\mathcal{P}_1$ consists of:

$$A \to a \qquad A \to \begin{matrix} a & A \end{matrix} \qquad A \to \begin{matrix} A \\ A \end{matrix}$$

$$B \to b \qquad B \to \begin{matrix} b & B \end{matrix} \qquad B \to \begin{matrix} B \\ B \end{matrix}$$

$$S_1 \to [C_{ij}]_{k+1,k+1} \quad \text{where } C_{ij} = \begin{cases} A & \text{if } i+j \text{ is even} \\ B & \text{otherwise} \end{cases}$$

$L(G_1, A)$, resp. $L(G_1, B)$ is the language of all non-empty pictures over $\{a\}$, resp. $\{b\}$, G_1 is a $CFPG(k+1)$ grammar generating $L(k+1)$.

The second statement will be proved by contradiction. Let $G = (V_T, V_N, S, \mathcal{P})$ be a $CFPG(k)$ grammar such that $L(G) = L_{ch}(k+1)$, n be an integer greater than $k \cdot (|\mathcal{P}| + 1)$. Without loss of generality, $\mathcal{P}$ does not contain productions of the form $C \to D$, where $C, D \in V_N$. Let L be the subset of $L_{ch}(k+1)$ containing exactly all square pictures of size n. At least $\lceil \frac{|L|}{|\mathcal{P}|} \rceil$ pictures in L can be generated by the same production in the first step. Let the production be $\Pi = S \to [A_{ij}]_{p,q}$ and the correspondent subset be L'. Without loss of generality, let $p \leq q$ which implies $q \geq 2$, since $\max(p, q) \geq 2$. Furthermore, we have $p, q \leq k$.

Let us divide pictures in L into groups with respect to vertical coordinates of borders between rectangles of a's and b's. More precisely, if P is a picture in L, which can be written as $\bigoplus [P_{ij}]_{k+1,k+1}$, where all P_{ij} are the pictures corresponding to the fields of the chessboard as it was described in Example 10, then the sequence $\mathcal{V}(P) = v_1, \ldots, v_k$, where

- $v_1 = \text{rows}(P_{1,1})$
- $v_i = v_{i-1} + \text{rows}(P_{i,1})$ for each $i = 2, \ldots, k$

is assigned to P. We can derive there are $\binom{n-1}{k}$ different sequences assigned to pictures in L, since there are $n-1$ borders between adjacent rows of a picture in L and, to form one sequence, we always choose k of them. Let $\mathcal{M}_v$ denote the set of these sequences. For $V \in \mathcal{M}_v$, let $\mathcal{G}(V)$ be a set of pictures as follows

$$\mathcal{G}(V) = \{P \mid P \in L \wedge \mathcal{V}(P) = V\}$$

Analogously, we can define sequences corresponding to horizontal coordinates of borders. Let $\mathcal{M}_h$ denote the set of these sequences. It holds $|\mathcal{M}_h| = \binom{n-1}{k}$ again. Furthermore, there is one to one correspondence among pictures in L and elements in $\mathcal{M}_v \times \mathcal{M}_h$, since each picture in L is determined by vertical and horizontal coordinates of the borders. It means

$$|L| = \binom{n-1}{k}^2$$

Moreover, for each $V \in \mathcal{M}_v$, $|\mathcal{G}(V)| = \binom{n-1}{k}$.

Now, we will consider another division of pictures in L' into groups, but this time with respect to how they are generated applying Π. A picture Q in L' can

be written as $\bigoplus[Q_{ij}]_{p,q}$, where $Q_{ij} \in L(G, A_{ij})$ (there are possibly more options how to generate Q, in this case we choose anyone of them). Now, the sequence $\mathcal{D}(Q)$ assigned to Q is the empty sequence if $p = 1$, else $\mathcal{D}(Q) = w_1, \ldots, w_{p-1}$, where

- $w_1 = \text{rows}(Q_{1,1})$
- $w_i = w_{i-1} + \text{rows}(Q_{i,1})$ for each $i = 2, \ldots, p-1$

There are $\binom{n-1}{p-1}$ such sequences. Let the set of these sequences be denoted by $\mathcal{M}_d$. For $D' \in \mathcal{M}_d$, let $\mathcal{G}(D') = \{P \mid P \in L' \wedge \mathcal{G}(P) = D'\}$. There must be $D \in \mathcal{M}_d$ such that

$$|\mathcal{G}(D)| \geq \frac{|L'|}{\binom{n-1}{p-1}}$$

We derive that this number is greater than $|\mathcal{G}(V)| = \binom{n-1}{k}$ (for any $V \in \mathcal{M}_v$).

First of all, the assumption $n > k \cdot (|\mathcal{P}| + 1)$ ensures that $n > 2 \cdot k$, since $\mathcal{P}$ cannot be empty. We have

$$p - 1 \leq k - 1 \leq \left\lfloor \frac{n-1}{2} \right\rfloor$$

which implies

$$\binom{n-1}{p-1} \leq \binom{n-1}{k-1} \leq \binom{n-1}{\lfloor \frac{n-1}{2} \rfloor}$$

hence

$$|\mathcal{G}(D)| \geq \frac{|L'|}{\binom{n-1}{k-1}}$$

Using the assumption on n, we can derive

$$n > k \cdot (|\mathcal{P}| + 1)$$

$$n > k + k \cdot |\mathcal{P}|$$

$$n - k > k \cdot |\mathcal{P}|$$

$$(n-k) \cdot \frac{(n-1)!}{(k-1)!(n-1-k)!} > |\mathcal{P}| \cdot k \cdot \frac{(n-1)!}{(k-1)!(n-1-k)!}$$

$$\frac{(n-1)!}{k!(n-1-k)!} > |\mathcal{P}| \cdot \frac{(n-1)!}{(k-1)!(n-k)!}$$

$$\binom{n-1}{k} > |\mathcal{P}| \cdot \binom{n-1}{k-1}$$

This inequality allows us to derive the desired result:

$$|\mathcal{G}(D)| \geq \frac{|L'|}{\binom{n-1}{p-1}} \geq \frac{|L'|}{\binom{n-1}{k-1}} \geq \frac{|L|}{|\mathcal{P}| \cdot \binom{n-1}{k-1}} > \frac{|L|}{\binom{n-1}{k}} = \binom{n-1}{k} = |\mathcal{G}(V)|$$

$|\mathcal{G}(D)| > |\mathcal{G}(V)|$ ensures there are two pictures P, P' in $\mathcal{G}(D)$ such that $\mathcal{V}(P) \neq \mathcal{V}(P')$. The pictures can be written in the form from Example 10: $P = \bigoplus[P_{ij}]_{p,q}$, $P' = \bigoplus[P'_{ij}]_{p,q}$. If we construct a new picture $O = [O_{ij}]_{p,q}$, where

- $O_{i,1} = P_{i,1}$ for all $i = 1, \ldots, p$
- $O_{i,j} = P'_{i,j}$ otherwise

then O is generated by G. It is a contradiction since the borders between a's and b's in the first column of O are not identical to the borders in the last column. □

For every one-dimensional context-free grammar, it is possible to construct an equivalent grammar in the Chomsky normal form, where productions are of the form $N \to AB$, $N \to a$ and $N \to \lambda$ (A, B, N are non-terminals, a is a terminal). Theorem 16 says that an analogy to the Chomsky normal form cannot be found for CFP grammars.

Of course, for a $CFPG$ $G = (V_T, V_N, S, \mathcal{P})$, we can construct an equivalent grammar, where terminals are present in productions right-hand sides only if the right-hand side is of size 1×1. Assuming $V_T = \{a_1, \ldots, a_k\}$, for each a_i, we can add a new non-terminal T_i and the production $a_i \to T_i$. Moreover, each occurrence of a_i in the right-hand side of a production in $\mathcal{P}$ of size different to 1×1 can be replaced by T_i. This normal form is presented in [9], however, we do not consider such a form to be an analogy to the Chomsky normal form, since sizes of the productions right-hand sides are not limited by a global constant as they are in the one-dimensional case.

7.5 Sentential Forms

Definition 16 *Let $G = (V_T, V_N, S, P)$ be a $CFPG$.* A Sentential form *over G is every triple $U = (B_H, B_V, R)$, where*

1. *B_V, B_H are finite, ascending sequences of real numbers in $\langle 0, 1\rangle$, $|B_V|, |B_H| \geq 2$. R is a finite set of tuples (x_1, x_2, y_1, y_2, X), where $X \in V_T \cup V_N$, $x_1, x_2 \in B_H$, $y_1, y_2 \in B_V$, $x_1 < x_2$, $y_1 < y_2$.*
2. *If $(x_1, x_2, y_1, y_2, X) \in R$ and $X \in V_T$ then there is not any $x \in B_V$ such that $x_1 < x < x_2$, neither any $y \in B_H$ such that $y_1 < y < y_2$.*
3. *Let the elements of B_V, resp. B_H be $v_1 < v_2 < \ldots < v_r$, resp. $h_1 < h_2 < \ldots < h_s$. For each pair $i \in \{1, \ldots, r-1\}$, $j \in \{1, \ldots, s-1\}$, there is exactly one element (x_1, x_2, y_1, y_2, X) in R such that $x_1 \leq v_i$, $v_{i+1} \leq x_2$, $y_1 \leq h_j$, $h_{j+1} \leq y_2$.*
4. *If $x \in B_H$, resp. $y \in B_V$, then there is an element (x_1, x_2, y_1, y_2, N) in R, such that $x = x_1$ or $x = x_2$, resp. $y = y_1$ or $y = y_2$.*

Informally, a sentential form U is a grid covered by labelled rectangles. The grid is determined by elements of B_V (vertical coordinates) and B_H (horizontal coordinates), rectangles correspond to elements in R – see Figure 7.15.

Point 3) says that each single element of the grid is covered by a rectangle. If a rectangle is labelled by a terminal, it covers exactly one grid's element (it is stated by point 2)). Point 4) says that each line of the grid has to form a border of a rectangle in R – no additional coordinates are allowed.

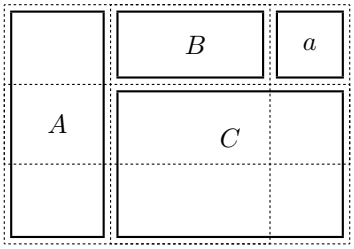


Figure 7.15: Sentential form (B_H, B_V, R), where the set of vertical, resp. horizontal coordinates is $B_V = (v_1, v_2, v_3, v_4)$, resp. $B_H = (h_1, h_2, h_3, h_4)$ and $R = ((h_1, h_4, v_1, v_2, A), (h_1, h_2, v_2, v_3, B), (h_2, h_4, v_2, v_4, C), (h_1, h_2, v_3, v_4, a))$.

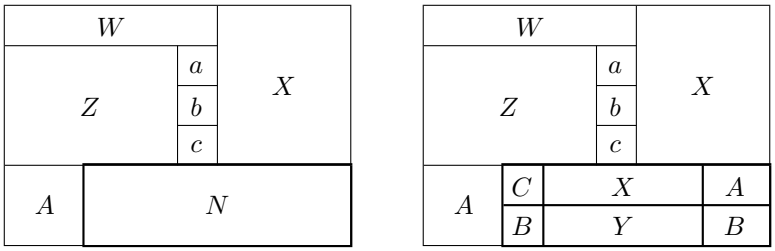


Figure 7.16: Sentential forms U and V. Assuming $N \to \begin{smallmatrix} CXA \\ BYB \end{smallmatrix}$ is a production, then U directly derives V ($U \Rightarrow V$).

Definition 17 *Let $G = (V_T, V_N, S, \mathcal{P})$ be a CFPG and $U = (B_H, B_V, R)$, $U' = (B'_H, B'_V, R')$ two sentential forms over G. Let the elements of B_V, resp. B_H be $v_1 < v_2 < \ldots < v_r$, resp. $h_1 < h_2 < \ldots < h_s$. We write $U \Rightarrow_G U'$ and say that U* directly derives *U' in grammar G iff there is a production $\Pi = N \to [A_{ij}]_{p,q}$ in $\mathcal{P}$ such that R' can be written as $R \setminus \{(x_1, x_2, y_1, y_2, N)\} \cup R_1$, where $(x_1, x_2, y_1, y_2, N) \in R$ and R_1 consists of $p \cdot q$ elements: for each pair $i = 1, \ldots, p$; $j = 1, \ldots, q$ it holds $(x_{ij}, x_{i+1,j}, y_{ij}, y_{i,j+1}, A_{ij}) \in R_1$, where x_{ij}'s, resp. y_{ij}'s are elements in B'_H, resp. B'_V satisfying*

$$x_1 = x_{1,j} < x_{2,j} < \ldots < x_{p,j} < x_{p+1,j} = x_2$$

$$y_1 = y_{i,1} < y_{i,2} < \ldots < y_{i,q} < y_{i,q+1} = y_2$$

Figure 7.16 shows an example of a direct derivation.

Definition 18 *Let $G = (V_T, V_N, S, \mathcal{P})$ be a CFPG and $U = (B_H, B_V, R)$, $U' = (B'_H, B'_V, R')$ two sentential forms over G. We say that U* derives *U' in G (and denote this relation by $U \Rightarrow^*_G U'$) iff $U = U'$ or there is a finite, non-empty sequence $\mathcal{S} = \{U_i\}_{i=0}^n$ of sentential forms over G such that*

$$U = U_0 \Rightarrow_G U_1 \Rightarrow_G U_2 \Rightarrow_G \ldots \Rightarrow_G U_{n-1} \Rightarrow_G U_n = U'$$

If it is clear which grammar G we refer to, we write $U \Rightarrow V$ instead of $U \Rightarrow_G V$ and $U \Rightarrow^* V$ instead of $U \Rightarrow^*_G V$. Let us consider a $CFPG$ $G = (V_T, V_N, S, \mathcal{P})$ and a sentential form $U = (B_V, B_H, R)$ over G, where, for all $(x_1, x_2, y_1, y_2, X) \in$

R, it is $X \in V_T$. Let the elements of B_V, resp. B_H be $v_1 < v_2 < \ldots < v_r$, resp. $h_1 < h_2 < \ldots < h_s$. We can interpret U as a picture over V_T. By the second point in Definition 16, for every pair $i = 1, \ldots, r-1$, $j = 1, \ldots, s-1$, there is in R exactly one element of the form $(v_i, v_{i+1}, h_j, h_{j+1}, X_{ij})$, thus we can assign to U the picture $\sigma(U)$ of size $(r-1) \times (s-1)$, where $\sigma(U)(i,j) = X_{ij}$. In the following text, we consider function σ to be defined on the set of all sentential forms of the described form.

Lemma 11 *Let G be a CFP grammar and $U = (B_H, B_V, R)$, $U' = (B'_H, B'_V, R')$ two sentential forms over G such that $U \Rightarrow U'$.*

In addition, let A_V, A_H, A'_V, A'_H be ascending, finite sequences of real numbers in $\langle 0, 1\rangle$, where $|A_V| = |B_V|$, $|A'_V| = |B'_V|$, $|A_H| = |B_H|$, $|A'_H| = |B'_H|$, $A_V \subseteq A'_V$ and $A_H \subseteq A'_H$. Let S be the set obtained from R by replacing each occurrence of the i-th element in B_V, resp. B_H by the i-th element in A_V, resp. A_H (i.e. each $(x_i, x_j, y_k, y_l, A) \in R$ is replaced by $(x'_i, x'_j, y'_k, y'_l, A)$, where x_t, resp. y_t, resp. x'_t, resp. y'_t is the t-th element of B_V, resp. B_H, resp. A_V, resp. A_H). Let S' be the set obtained in the same way using sets R', A'_V and A'_H.

Then, $V = (A_V, A_H, S)$ and $V' = (A'_V, A'_H, S')$ are sentential forms and $V \Rightarrow V'$. Moreover, if $\sigma(U')$ is defined then $\sigma(V')$ is defined too and $\sigma(U') = \sigma(V')$.

Proof. Follows obviously from the definition of sentential forms and direct derivation. U and V, resp. U' and V' differ just in positions of particular coordinates, but the structure of the sentential forms remains the same. □

Let us consider a CFP grammar $G = (V_T, V_N, \mathcal{P}, S_0)$ and two sentential forms $U = (B_H, B_V, R)$, $V = (A_H, A_V, T)$ over G. Furthermore, let $r = (x_1, x_2, y_1, y_2)$ be a tuple of elements in $\langle 0, 1\rangle$, such that $x_1 < x_2$ and $y_1 < y_2$. We say that V is *a sub-form* of U *given by* r if the following conditions are fulfilled:

1. $A_H \subseteq B_H \cap \langle x_1, x_2\rangle$, $A_V \subseteq B_V \cap \langle y_1, y_2\rangle$
2. $(x_a, x_b, y_a, y_b, N) \in T$ iff $(x_a, x_b, y_a, y_b, N) \in R$, $x_1 \leq x_a$, $x_b \leq x_2$, $y_1 \leq y_a$ and $y_b \leq y_2$

It is evident that for a pair U and r, there is at most one sub-form of U given by r. If the sub-form exists, we will denote it $s(U, r)$. To give an example, let us take some element $(x_1, x_2, y_1, y_2, N) \in R$ and $r = (x_1, x_2, y_1, y_2)$. Then, $s(U, r)$ exists (is defined) – it consists of a non-terminal or terminal covering the whole grid. Formally

$$s(U, r) = \{\{x_1, x_2\}, \{y_1, y_2\}, \{(x_1, x_2, y_1, y_2, N)\}\}$$

On the other hand, if $\langle x_1, x_2\rangle \cap B_H = \emptyset$, then $s(U, r)$ is not defined.

Lemma 12 *Let $G = (V_T, V_N, \mathcal{P}, S_0)$ be a CFP grammar, $U = (B_H, B_V, R)$, $V = (A_H, A_V, T)$ sentential forms over G such that $U \Rightarrow^* V$. Let $r = (x_1, x_2, y_1, y_2)$ be a quadruple of real numbers in $\langle 0, 1\rangle$.*

1. *If the sub-form $s(U, r)$ is defined, then $s(V, r)$ is defined as well and $s(U, r) \Rightarrow^* s(V, r)$.*

2. *If V can be derived from U in n steps ($n \in \mathbb{N}^+$), i.e. if there are $n-1$ suitable sentential forms $U_1, \ldots, U_{n-1}$ such that*

$$U \Rightarrow U_1 \Rightarrow U_2 \Rightarrow \ldots \Rightarrow U_{n-1} \Rightarrow V$$

and $s(U,r)$ is defined, then it is possible to derive $s(V,r)$ from $s(U,r)$ in n or less steps.

Proof. If $U = V$, the first statement is fulfilled trivially. Let $U \neq V$ and $U = U_0 \Rightarrow U_1 \Rightarrow \ldots \Rightarrow U_n = V$ be a derivation of V from U in G. Since $s(U,r)$ is defined, it is evident that each $S_i = s(U_i, r)$ is also defined. Moreover, for every S_i, S_{i+1} $(i = 0, \ldots, n-1)$ we have $S_i = S_{i+1}$ or $S_i \Rightarrow S_{i+1}$. That proves the lemma. □

Definition 19 *Let $G = (V_T, V_N, S, \mathcal{P})$ be a CFP grammar. We say that a picture O over V_T is generated by a non-terminal $N \in V_N$ in G* via sentential forms *iff there is a sentential form U such that*

$$(\{0,1\}, \{0,1\}, \{(0,0,1,1,N)\}) \Rightarrow^* U$$

and $\sigma(U) = O$. $L_S(G,N)$ denotes the language consisting of all pictures that can be generated from N in G via sentential forms. $L_S(G) = L_S(G,S)$ is the language generated by G via sentential forms.

Let $\Pi = N \to [A_{ij}]_{p,q}$ be a production. To simplify the proof of the next theorem, we define a function assigning a sentential form to a production as follows: $\mathrm{sf}(\Pi) = (B_V, B_H, R)$, where $B_V = \{\frac{i}{p} \mid i = 0, \ldots, p\}$, $B_H = \{\frac{j}{q} \mid j = 0, \ldots, q\}$ and $R = \{(\frac{i}{p}, \frac{i+1}{p}, \frac{j}{q}, \frac{j+1}{q}, A_{ij}) \mid i = 0, \ldots, p-1; j = 0, \ldots, q-1\}$.

Theorem 17 *For every CFP grammar G, $L(G) = L_S(G)$.*

Proof. Let $G = (V_T, V_N, S, \mathcal{P})$ be a CFP grammar.

1) Firstly, we show $L(G,N) \subseteq L_S(G,N)$ for an arbitrary $N \in V_N$. Let $O = [a_{ij}]_{p,q}$ be a picture in $L(G,N)$, T be a derivation tree for O in G. We prove by induction on the depth of T that $O \in L_S(G,N)$.

If the depth of T is 1 then there is a production $\Pi = N \to O$ in $\mathcal{P}$. We have $(\{0,1\}, \{0,1\}, \{(0,0,1,1,N)\}) \Rightarrow \mathrm{sf}(\Pi)$ and $\sigma(\mathrm{sf}(\Pi)) = O$, thus $O \in L_S(G,N)$.

Let the depth of T be $k > 1$, $\Pi = N \to [A_{ij}]_{p,q}$ be the production assigned to the root of T, v_{ij} be the direct descendant of the root of T labelled by A_{ij} and $T(v_{ij})$ be the subtree of T having v_{ij} as the root. O can be written as $\bigoplus [O_{ij}]_{p,q}$, where each O_{ij} is in $L(G, A_{ij})$. The depth of each $T(v_{ij})$ is at most $k-1$. It means, after applying the hypothesis, if $A_{ij} \in V_N$, then there is a sentential form U_{ij} such that $\sigma(U_{ij}) = O_{ij}$ and $(\{0,1\}, \{0,1\}, \{(0,0,1,1,A_{ij})\}) \Rightarrow^* U_{ij}$. O can be derived as follows:

1. $(\{0,1\}, \{0,1\}, \{(0,0,1,1,S)\}) \Rightarrow \mathrm{sf}(\Pi)$
2. Let $\mathrm{sf}(\Pi) = (B_V, B_H, R)$. For each element $(x_{ij}, \overline{x}_{ij}, y_{ij}, \overline{y}_{ij}, A_{ij})$ in R, where $A_{ij} \in V_N$, a sequence of derivations can be applied on $s(\mathrm{sf}(\Pi), (x_{ij}, \overline{x}_{ij}, y_{ij}, \overline{y}_{ij}))$ to derive U'_{ij} such that $\sigma(U'_{ij}) = \sigma(U_{ij})$. It is ensured by Lemma 11. It should be clear that all these sequences of derivations can be adjusted and applied on $\mathrm{sf}(\Pi)$ to derive U such that $\sigma(U) = O$.

2) We prove $L_S(G,N) \subseteq L(G,N)$. Let O be a picture in $L_S(G,N)$. By induction on the length of a derivation of O: Let $U_0 = (\{0,1\},\{0,1\},\{(0,0,1,1,S)\}) \Rightarrow U$, where $\sigma(U) = O$. Then the considered grammar contains the production $S \to O$. A derivation tree for O of the depth one can be easily constructed.

Let $U_0 \Rightarrow U_1 \Rightarrow \ldots \Rightarrow U_n$, where $n \geq 2$ and $\sigma(U_n) = O$. U_1 is of the form (B_H, B_V, R), where

- $B_H = \{h_i\}_{i=0}^{p}$ and $0 = h_0 < h_1 < \ldots < h_p = 1$
- $B_V = \{v_i\}_{i=0}^{q}$ and $0 = v_0 < v_1 < \ldots < v_q = 1$
- $R = \{(v_i, v_{i+1}, h_i, h_{i+1}, A_{ij}) \,|\, i = 0,\ldots,p-1;\ j = 0\ldots,q-1\}$, where $S \to [A_{ij}]_{p,q}$ is a production in $\mathcal{P}$

For $i \in \{1,\ldots,p\}$ and $j \in \{1,\ldots,q\}$, let $s(i,j)$, resp. $s'(i,j)$ denote $s(U_1,(v_{i-1},v_i,h_{j-1},h_j))$, resp. $s(U_n,(v_{i-1},v_i,h_{j-1},h_j))$. Then, by Lemma 12, for any possible pair i, j

$$s(i,j) \Rightarrow^* s'(i,j)$$

Since $O = \sigma(U_n)$, we have

$$O = \bigoplus [\sigma(s'(i,j))]_{p,q}$$

If we denote each sub-picture $\sigma(s'(i,j))$ by $O_{i,j}$, then $O_{i,j} \in L_S(G,A_{i,j})$ – Lemma 11 can be used to obtain a proper derivation. Moreover, the length of this derivation is at most $n-1$ (Lemma 12), thus $O_{i,j} \in L(G,A_{i,j})$. Now, it is evident that a derivation tree proving $O \in L(G,S)$ can be constructed.

□

7.6 CFP Grammars over One-symbol Alphabet, Pumping Lemma

In this section, we study which integral functions are representable by CFP languages (as for the representation of functions by languages, see Definition 5).

In our descriptions, when we write productions, right-hand sides can contain elements of the form A^k, where A is a terminal or non-terminal and $k \in \mathbb{N}^+$. This notation abbreviates k repetitions of A in the related row. For example

$$N \to \begin{matrix} A^3 & B \\ C^3 & D \end{matrix}$$

should be interpreted as

$$N \to \begin{matrix} A & A & A & B \\ C & C & C & D \end{matrix}$$

Proposition 13 *Every integral constant function $f : \mathbb{N}^+ \to \{c\}$, where $c \geq 1$, is representable by a CFP language.*

Proof. It is evident that the following CFP grammar G generates a language representing f: $G = (\{a\}, \{S, C\}, \mathcal{P}, S)$, $\mathcal{P}$ consists of the productions

$$C \to a^c \qquad S \to C \qquad S \to \begin{matrix} C \\ S \end{matrix}$$

□

Proposition 14 *Let f be a function representable by a CFP language and $c \in \mathbb{N}^+$. Then, $c \cdot f$ is representable by a CFP language as well.*

Proof. Let L be a CFP language that represents f, $G = (V_T, V_N, \mathcal{P}, S)$ be a CFP grammar generating L and $S_0 \notin V_N$ a non-terminal. If we define

$$G' = (V_T, V_N \cup \{S_0\}, \mathcal{P} \cup \{S_0 \to S^c\}, S_0)$$

then, G' is a CFP grammar and $L(G')$ represents $c \cdot f$. Whenever an occurrence of S in $S_0 \to S^c$ is substituted by a picture P, it forces other pictures substituted to the other occurrences of S to have the same number of rows as P. The only possibility is to choose P in all these cases again. □

Proposition 15 *Let f, g be two functions representable by CFP languages. Then, $f + g$ is representable by a CFP language as well.*

Proof. Let L_1, resp. L_2 be a CFP language representing f, resp. g, $G_1 = (V_T, V_1, \mathcal{P}_1, S_1)$, resp. $G_2 = (V_T, V_2, \mathcal{P}_2, S_2)$ be a CFP grammar generating L_1, resp. L_2. Without loss of generality, we can assume $V_1 \cap V_2 = \emptyset$. For a new non-terminal $S_0 \notin V_1 \cup V_2$, we can define the following CFP grammar

$$G = (V_T, V_1 \cup V_2 \cup \{S_0\}, \mathcal{P}_1 \cup \mathcal{P}_2 \cup \{S_0 \to S_1 S_2\}, S_0)$$

It is easy to see that $L(G)$ represents $f + g$. □

Proposition 16 *Let $f : \mathbb{N}^+ \to \mathbb{N}^+$ be a function given by the following recurrent formula:*

$$f(1) = c$$

$$\forall n \geq 2 : \; f(n+1) = k \cdot f(n) + g(n)$$

where g is a function representable by a CFP language or $g(n) = 0$ for all $n \in \mathbb{N}^+$ and c, k are positive integers. Then, f is representable by a CFP language.

Proof. If $g \neq 0$, let $G_1 = (\{a\}, V_1, \mathcal{P}_1, S_1)$ be a CFP grammar generating the language over $\{a\}$ that represents g. We can construct grammar G, such that $L(G)$ represents f, as follows. $G = (\{a\}, V_1 \cup \{S_0, A\}, \mathcal{P}_1 \cup \mathcal{P}, S_0)$, where S_0 and A are both non-terminals, not contained in V_1, and $\mathcal{P}$ consists of the following productions:

$$(1)\ A \to a \qquad (2)\ A \to \begin{matrix} a & A \end{matrix} \qquad (3)\ S_0 \to a^c \qquad (4)\ S_0 \to \begin{matrix} S_0^k & S_1 \\ A^k & A \end{matrix}$$

We show that $L(G)$ represents f by induction on the number of rows of a picture in L. Firstly, $L(G, A)$ contains exactly all pictures of the form a^k, where k is any positive integer. Next, there is exactly one picture in $L(G)$ that contains one row. It is generated in the first step using production (3) (pictures generated using production (4) in some step have at least two rows), thus the picture equals to $[1, c]$. Let $n > 1$ be an integer and let us assume, for each integer m, $1 \leq m < n$, L contains exactly one picture having m rows and this picture equals $[m, f(m)]$. Any picture having n rows must be generated by production (4) in the first step. All pictures in $L(G, A)$ consist of one row, hence S_0 and S_1 on the right-hand side of the production have to be substituted by pictures having $n-1$ rows. The only possibility is to use $[n-1, f(n-1)]$ and $[n-1, g(n-1)]$, which implies the picture $[n, k \cdot f(n-1) + g(n-1)]$ is in $L(G)$.

If $g = 0$, it is sufficient to take $G = (\{a\}, \{S_0, A\}, \mathcal{P}', S_0)$, where $\mathcal{P}'$ consists of productions (1), (2), (3) and production (5):

$$(5)\ S_0 \rightarrow \begin{matrix} S_0^k \\ A^k \end{matrix}$$

□

Let us use Proposition 16 to obtain examples of functions representable by a CFP language.

If we take $c = k = 1$ and $g(n) = 1$ for all $n \in \mathbb{N}^+$, we get $f(n) = n$ is representable. $f(n) = n^2$ is representable as well, since

$$f(n+1) = (n+1)^2 = n^2 + 2 \cdot n + 1 = f(n) + 2 \cdot n + 1$$

thus it is sufficient to choose $c = k = 1$, $g(n) = 2 \cdot n + 1$. Note that $g(n) = 2 \cdot n + 1$ is representable. It can be written as $g(n) = f(n) + f(n) + 1$. Since all summands are representable functions, their sum is representable by Proposition 15.

In general, we can conclude that if f is a polynomial function

$$f(n) = \sum_{i=0}^{k} n^i \cdot a_i$$

where a_i's are natural numbers and at least one of them is positive, then f is representable by a CFP language. It can be shown by induction on the degree of polynomial using the formula

$$(n+1)^k = \sum_{i=0}^{k} \binom{n}{i} \cdot n^i = n^k + \sum_{i=0}^{k-1} \binom{n}{i} \cdot n^i$$

Another examples are exponential functions. Let us choose $c = 1$, $g(n) = 0$ for all $n \in \mathbb{N}^+$ and let k be a parameter. Proposition 16 gives $f(n) = k^n$ is representable by a CFP language.

We have already shown that the language $L = \{[n, n^2] \mid n \in \mathbb{N}^+\}$ is not recognizable by a FSA (it is a consequence of Theorem 1). On the other hand, L represents $f(n) = n^2$ and it is a CFP language. It can be given as another example of a language that is in $L(CFPG)$ and is not in $L(FSA)$ (see Theorem 13).

The next result we are going to present is a contribution to the study of the question whether there is some sort of the pumping lemma applicable on CFP languages. We show that it is possible to "pump" pictures that are sufficiently "wide" or "high" respectively.

We recall the pumping lemma (also known as $uvwxy$ theorem) from the classical theory of languages, which says: For every (one-dimensional) context-free language L, there are constants c_1, c_2 such that if α is a word in L such that $|\alpha| \geq c_2$, then it can be written as $uvwxy$, where $|vwx| \leq c_1$, $|v| + |x| \geq 1$ and, for any natural number i, $uv^iwx^iy \in L$.

Still considering the one-dimensional theory, the lemma can be extended on sentential forms in the following way: Let $G = (V_T, V_N, P, S_0)$ be a one-dimensional context-free grammar. There are constants c_1, c_2, such that if $N \Rightarrow_G^* \beta$, where N is a non-terminal, β a sentential form of length at least c_1, then β can be written as $uvwxy$, where $N \Rightarrow_G^* uv^iwx^iy$ for each $i \in \mathbb{N}$.

Theorem 18 *Let L be a CFP language over the alphabet $\Sigma = \{a\}$. There is a positive integral constant c for L, such that if $[m, n] \in L$ and $n \geq c^m$, then, for all $i \in \mathbb{N}$, $[m, n + i \cdot n!] \in L$ as well.*

Proof. Let $G = (\Sigma, V_N, \mathcal{P}, S_0)$ be a CFP grammar generating L. Without loss of generality we will assume, P does not contain productions of the form $A \to B$, where A, B are non-terminals in V_N. Let p be the maximal number of columns forming the right-hand side of a production in P, i.e.

$$p = \max\{\mathrm{cols}(\mathcal{A}) \,|\, N \to \mathcal{A} \in \mathcal{P}\}$$

Let $G' = (\Sigma, V_N, \mathcal{P}', S_0)$ be a one-dimensional context-free grammar, where

$$\mathcal{P}' = \{N \to \mathcal{A} \,|\, N \to \mathcal{A} \in \mathcal{P} \wedge \mathrm{rows}(\mathcal{A}) = 1\}$$

i.e. G' is created based on G by taking all productions having one-row right-hand sides. Let d be a constant given by the extended pumping lemma applied on G' determining the required length of sentential forms. It means, if $N \Rightarrow_{G'}^* \beta$ and $|\beta| \geq d$, then β can be written as $uvwxy$, etc. We put $c = p \cdot d$.

For any $N \in V_N$ and an integer $m \geq 1$, we define auxiliary languages $L(N, m)$:

$$L(N, m) = \{O \,|\, O \in L(G, N) \wedge \mathrm{rows}(O) = m \wedge \mathrm{cols}(O) \geq c^m\}$$

We are ready to prove the theorem by induction on m (the number of rows a of picture). Let O be a picture in $L(N, 1)$. Since $\mathrm{rows}(O) = 1$ and $n = \mathrm{cols}(O) \geq p \cdot d \geq d$, the one-dimensional pumping lemma can be applied on O implying $[1, n + i \cdot k] \in L(N, 1)$ for some suitable integer $1 \leq k \leq n$ and each natural number i, thus $[1, n + i \cdot n!] \in L(N, 1)$ for all $i \in \mathbb{N}$.

Let $m > 1$ and let us suppose that, for each integer m', $1 \leq m' < m$ and each nonterminal $N \in V_N$, it holds that $[m', n] \in L(N, m')$ implies $[m', n + i \cdot n!]$ for all $i \in \mathbb{N}$. We distinguish two cases depending on how a picture $O \in L(N, m)$ is generated in the first step. Let $n = \mathrm{cols}(O)$. If it is possible to generate O in the first step by a production $\Pi = N \to [A_{ij}]_{r,s}$, where $r > 1$, then O can be written as $\bigoplus [U_{ij}]_{r,s}$, where $U_{ij} \in L(G, A_{ij})$ if A_{ij} is a non-terminal and

$U_{ij} = A_{ij}$ if A_{ij} is a terminal. Since $s \leq p$, there is a column of index s_0 such that, for each $i = 1, \ldots, r$

$$n_0 = \mathrm{cols}(U_{i,s_0}) \geq \frac{n}{p}$$

We have

$$n \geq (p \cdot d)^m$$

thus

$$n_0 \geq \frac{n}{p} \geq (p \cdot d)^{m-1} \cdot d \geq (p \cdot d)^{m-1}$$

Since $\mathrm{rows}(U_{i,s_0}) \leq m - 1$ it holds $n_0 \geq c^{\mathrm{rows}(U_{i,s_0})}$. It means, each U_{i,s_0} is in $L(A_{i,s_0}, \mathrm{rows}(U_{i,s_0}))$, thus, with respect to our inductive hypothesis, $U_{i,s_0} = [r_i, n_1]$ can be "pumped" to $U'_{i,s_0}(k) = [r_i, n_1 + k \cdot n_1!] \in L(G, A_{i,s_0})$ for all $k \in \mathbb{N}$ (note that it would be also sufficient if $n_0 \geq n/(p \cdot d)$ only – this will be used in the second part of the proof). If we replace each sub-picture U_{i,s_0} in O by $U'_{i,s_0}(k)$ for a fixed number k, we get $[m, n + k \cdot n_1!] \in L(N, m)$. Since $n_1!$ divides $n!$, we are finished in this case.

If there is no possibility to generate O by a production having at least two rows on the right-hand side in the first step, we have to consider a production with the right-hand side consisting of one row, i.e. $\Pi = N \to A_1 \ldots A_s$. O can be written as $U_1 \oplus U_2 \oplus \ldots \oplus U_s$ in this case. One of U_i's must contain at least $\frac{n}{p}$ columns again. Let U_k be such a sub-picture. Once again, we examine, which production is used to generate U_k from A_k in the first step. If the production has more than one row, we can follow the steps we have already described before (in this case, we have $n_0 \geq n/(p \cdot d)$, which is still sufficient). Otherwise U_k can be expressed as $V_1 \oplus V_2 \oplus \ldots \oplus V_t$ for suitable pictures V_i. It means, O can be written as a column concatenation of $s + t - 1$ sub-pictures:

$$O = U_1 \oplus \ldots \oplus U_{k-1} \oplus V_1 \oplus \ldots \oplus V_t \oplus U_{k+1} \oplus \ldots \oplus U_s$$

Now, it is possible to take a sub-picture consisting of at least $n/(s + t - 1)$ columns and repeat the steps described above until a sub-picture that can be generated by a production with more than one row is found or O is expressed as a concatenation of d or more sub-pictures.

In the first case the found sub-picture has at least n/d columns. Again, this number is sufficiently large to apply the steps related to a production with more than one row and finish the proof. The second case implies $N \Rightarrow^*_{G'} \beta$ and $\beta \Rightarrow^*_G W$, where β, W are sentential forms (we treat β as a two-dimensional sentential form in the second relation) and $\sigma(W) = O$. β is a one-dimensional sentential form of length at least d. It means, we can apply the pumping lemma on β, hence $[m, n + i \cdot k] \in L(N, m)$ for some $k \leq n$ and each $i \in \mathbb{N}$. Finally, $k | n!$, thus each $[m, n + i \cdot n!]$ is in $L(N, m)$ as well. □

Remarks on the theorem follows:

1. To achieve a good readability of the proof we have worked with grammars over the one-symbol alphabet $\Sigma = \{a\}$ only, however, the proof can be easily extended on grammars over any alphabet containing any number of symbols, since the proof does not depend on Σ.

0	1	0	0	0	0	0
0	0	0	1	0	0	0
1	0	0	0	0	0	0
0	0	0	0	0	0	1
0	0	0	0	1	0	0
0	0	1	0	0	0	0
0	0	0	0	0	1	0

Figure 7.17: Example of a picture in L_R.

2. The theorem is also valid if we swap the terms 'row' and 'column', i.e. if $[m, n] \in L$ and $m \geq c^n$, then $[m + i \cdot m!, n] \in L$.

3. The fact that exponential functions are representable by CFP languages indicates that the condition $n \geq d^m$ in Theorem 18 is optimal. It cannot be possible to "pump" languages $\{[m, c^m] \mid m \in \mathbb{N}^+\}$, where $c \in \mathbb{N}$, $c \geq 2$. This observation rises a question if some kind of the pumping lemma that pumps rows and columns together can be found. Then, it could also work for the mentioned languages.

For $c \in \mathbb{N}^+$, let $\exp_c : \mathbb{N}^+ \to \mathbb{N}^+$ be an integral function such that $\forall n \in \mathbb{N}^+ : \exp_c(n) = c^n$.

Proposition 17 *Let $f : \mathbb{N}^+ \to \mathbb{N}^+$ be a function satisfying $f = o(\exp_c)$ for each positive integer c. Then, f cannot be represented by a CFP language.*

Proof. By contradiction. Let f be a function such that $f = o(\exp_c)$ for all $c \in \mathbb{N}^+$ and L be a CFP language representing f. By Theorem 18, there is a constant d fulfilling: if some picture $[m, n]$ is in L and $n \geq d^m$, then $[m, n+i \cdot n!] \in L$ for each positive integer i. We have $f = o(\exp_d)$, thus it is possible to find m_0, such that $f(m_0) > d^{m_0}$. L represents f, hence $[m_0, f(m_0)] \in L$, which implies $[m_0, f(m_0) + f(m_0)!] \in L$. It contradicts the fact that L represents f. □

7.7 Closure Properties

Example 11 Let L_R be the language over the alphabet $\{0, 1\}$ consisting of all pictures having in every row and column exactly one symbol 1. Formally

$$L_R = \{O \mid O \in \{0,1\}^{**} \wedge \forall i = 1, \ldots, \text{rows}(O)\ \exists! j : O(i,j) = 1 \\ \wedge\ \forall j = 1, \ldots, \text{cols}(O)\ \exists! i : O(i,j) = 1\}$$

Figure 7.7 shows an example of a picture in L_R. Every picture in L_R is a square picture. The reason is that the number of occurrences of the symbol 1 in such a picture has to equal the number of rows of the picture as well as the number of columns.

A picture in L_R of size n can be interpreted as one of the possible placements of n rooks on the $n \times n$ chessboard, where no two rooks threat each other (rooks correspond to occurrences of the symbol 1).

Lemma 13 *Let r be a natural number and $\mathcal{S} = \{s_i\}_{i=1}^n$ a sequence of natural numbers, where $n \geq 2$ and, for all $i = 1, \ldots, n$: $s_i \leq r \leq s = \sum_{i=1}^n s_i$. Then*

$$\prod_{i=1}^{n} s_i! \leq r! \cdot (s-r)!$$

Proof. We can consider $\mathcal{S}$ to be sorted in the descending order, i.e. $s_1 \geq s_2 \geq \ldots \geq s_n$. The expressions on both sides of the inequality to be proved can be replaced by products of s numbers (each $m!$, where $m > 0$, is expressed by $1 \cdot 2 \cdot \ldots \cdot m$, eventual occurrences of 0! are omitted). It means, the left-hand side is:

$$1 \cdot 2 \ldots s_1 \cdot 1 \ldots s_2 \ldots 1 \ldots s_n$$

and the right-hand side:

$$1 \cdot 2 \ldots r \cdot 1 \ldots (s-r)$$

If these products are divided by $s_1! \cdot t!$, where $t = \min(s_2, s-r, r)$ and expressed as products of $s - s_1 - t$ numbers sorted in the ascending order (in our case, to divide one product by $m!$ means to remove m factors of values $1, \ldots, m$), then each i-th factor of the left-hand side is not greater than the i-th factor of the right-hand side. E.g., if $t = s - r$, then the left-hand side (written with non-sorted factors) is

$$(s-r+1) \cdot (s-r+2) \ldots s_2 \ldots 1 \ldots s_n \qquad (L)$$

while the right-hand side is

$$(s_1+1) \cdot (s_1+2) \ldots r \qquad (R)$$

The product $(s-r+1) \ldots s_2$ is not surely greater than the product $(s - s_2 + 1) \ldots r$, since $s_2 \leq r$ and both products consist of the same number of factors. If the remaining factors in (L) are sorted in the ascending order into a sequence $\mathcal{S}_L = \{a_i\}_{i=1}^k$ and remaining factors in (R) into $\mathcal{S}_R = \{b_i\}_{i=1}^k$, where $k = s - s_1 - t - s_2 + s - r$, then, assuming $k > 0$, we have

1. $\mathcal{S}_R$ is an ascending sequence, where $b_{i+1} > b_i$ for all $i \in 1, \ldots, k-1$ and $b_1 \geq 1$
2. $\mathcal{S}_L$ is a monotonic sequence, where $a_{i+1} = a_i$ or $a_{i+1} = a_i + 1$ for all $i = 1, \ldots, k-1$ and $a_1 = 1$

These two points imply that $a_i \leq b_i$ for each $i \in \{1, \ldots, k\}$.

The cases $t = r$ and $t = s_2$ can be inspected analogously. □

Lemma 14 $L_R \notin L(CFPG)$

Proof. By contradiction. Let $G = (V_N, V_T, \mathcal{P}, S)$ be a CFP grammar generating L_R. Without loss of generality, we can assume $\mathcal{P}$ does not contain any production of the form $A \to B$, where A, B are non-terminals. Let n be an integer, L_1 the set of all square pictures in L_R of size n. Let us assume n is large enough so that none of the pictures in L_1 equals the right-hand side of a production in $\mathcal{P}$. We have $|L_1| = n!$. At least $\lceil \frac{n!}{|P|} \rceil$ pictures are generated in

the first step by the same production. Let the production be $\Pi = S \to [A_{ij}]_{p,q}$. We denote by L_2 the set of pictures that can be derived by Π in the first step. Without loss of generality, $p \geq q$ and $p \geq 2$. Our goal will be to derive an upper bound on $|L_2|$ leading to a contradiction to $|L_2| \geq \lceil \frac{n!}{|P|} \rceil$.

We say that D is a *division* of a picture if it is a pair

$$((r_1, \ldots, r_{l_1}), (s_1, \ldots, s_{l_2}))$$

where $r_1 < \ldots < r_{l_1}$ and $s_1 < \ldots < s_{l_2}$ are integers.

For a division $D = ((r_1, \ldots, r_p), (s_1, \ldots, s_q))$, let $L(D)$ denote the subset of L_2 containing exactly each picture U that can be expressed as $U = \bigoplus [U_{ij}]_{p,q}$, where $U_{ij} \in L(G, A_{ij})$ and $\mathrm{rows}(U_{ij}) = r_i$, $\mathrm{cols}(U_{ij}) = s_j$ for all possible pairs of indices i, j.

Let us consider a pair of pictures U, V in $L(D)$, $U = \bigoplus [U_{ij}]_{p,q}$, $V = \bigoplus [V_{ij}]_{p,q}$, where for all $i = 1, \ldots, p$, $j = 1, \ldots, q$, $\mathrm{rows}(U_{ij}) = \mathrm{rows}(V_{ij}) = r_i$ and $\mathrm{cols}(U_{ij}) = \mathrm{cols}(V_{ij}) = s_j$. For an arbitrary pair of indices l, m, let U' be created from U by replacing U_{lm} by V_{lm}. Since U' is in L_2 again, U_{lm} and V_{lm} have to contain the same number of symbols 1. Moreover, if $i_1, \ldots, i_k$ are indices of all rows, resp. columns in U_{lm} containing 1 then they have to be equal to the analogous indices of rows, resp. columns in V_{lm} (if not, let, for example, the i_1-th row of U_{lm} contain 1 and the i_1-th row of V_{lm} do not contain 1. Then, U' has a row without the symbol 1).

Let k_{ij} be the number of occurrences of the symbol 1 in U_{ij}. There are $k_{ij}!$ different pictures (including U_{ij}) that can replace U_{ij} in U so that the created picture is in L_2 again. It means, the number of elements in $L(D)$ have to satisfy the following inequality:

$$|L(D)| \leq \prod_{i=1,\ldots,p; j=1,\ldots,q} k_{ij}! \tag{7.1}$$

Moreover, $|L_2| \leq \sum_D |L(D)|$, where the sum goes trough all possible divisions D. To obtain an upper bound on $|L_2|$, we distinguish two cases with respect to coordinates of a given division $D = ((r_1, \ldots, r_p), (s_1, \ldots, s_q))$:

1) $\max(r_1, \ldots, r_p) \geq n - 4p$ and $\max(s_1, \ldots, s_q) \geq n - 4p$; i.e., one of the pictures U_{ij} fulfils $\mathrm{rows}(U_{ij}) \geq n - 4p$ and $\mathrm{cols}(U_{ij}) \geq n - 4p$. Let us say that such a picture has property (B).

2) $\max(r_1, \ldots, r_p) < n - 4p$ or $\max(s_1, \ldots, s_q) < n - 4p$; i.e., each of the pictures U_{ij} is of width or height less than $n - 4p$.

We can assume n is large enough comparing to p so that, for $U = \bigoplus [U_{ij}]_{p,q}$, there is maximally one picture U_{ij} which has property (B).

Ad 1). First of all, we estimate how many different divisions contain a subpicture of property (B). In the sequence $r_1, \ldots, r_p$, there are p possibilities which of the elements to choose to be the maximal one. This element can be then of $3p+2$ different values maximally (possible values are $n-4p, \ldots, n-p+1$). Each of the remaining elements of the sequence can be of a value in $1, \ldots, 4p$, thus the number of all suitable sequences is bounded by

$$c_1{}' = p \cdot (3p + 2) \cdot (4p)^{p-1}$$

We can similarly derive that the number of all suitable sequences $s_1, \ldots, s_q$ is bounded by ${c_1}'' = q \cdot (4p - q + 2) \cdot (4p)^{q-1}$, thus the number of divisions in the case 1) is bounded by $c_1 = {c_1}' \cdot {c_1}''$.

Now, for a given division D, we derive an upper bound on $L(D)$. Without loss of generality, we can assume $U_{1,1}$ is the only picture which has property (B). $\max(\text{rows}(U_{1,1}), \text{cols}(U_{1,1})) \leq n - p + 1$ (each other U_{ij} is of size at least 1×1), hence $k_{1,1} \leq (n - p + 1)!$. Moreover, $k_{ij} \leq k_{1,1}$ for all pairs $i = 1, \ldots, p$; $j = 1, \ldots, q$. If we put $r = n - p + 1$, $s = n$ and apply Lemma 13 on values k_{ij} in (7.1), we derive

$$|L(D)| \leq (n - p + 1)! \cdot (p - 1)! \leq \frac{n!}{n} \cdot (p - 1)!$$

Ad 2). In this case, it is sufficient to consider the number of divisions to be bounded by n^{2p} (each element in both sequences can be of a value in $1, \ldots, n$). We apply Lemma 13 on values k_{ij} again. Since each $k_{ij} \leq n - 4p - 1$, we get

$$|L(D)| \leq (n - 4p - 1)! \cdot (4p + 1)!$$

We proceed by deriving an upper bound on the expression $n^{2p} \cdot (n - 4p - 1)!$

$$\begin{aligned}(n - 4p - 1)! \cdot n^{2p} &= \frac{n!}{n \cdot (n-1) \ldots (n - 4p)} \cdot n^{2p} \\ &\leq \frac{n! \cdot n^{2p}}{n \cdot (n - 4p)^{4p}} = \frac{n!}{n} \cdot \left(\frac{n}{(n - 4p)^2} \right)^{2p}\end{aligned}$$

Finally, we can put together results of both cases and estimate the sum $\sum_D |L(D)|$

$$|L_2| \leq \sum_D |L(D)| \leq c_1 \cdot (p - 1)! \cdot \frac{n!}{n} + (4p + 1)! \cdot \frac{n!}{n} \cdot \left(\frac{n}{(n - 4p)^2} \right)^{2p} = f(n)$$

It is evident that, for sufficiently large n (with respect to p), we have $|L_2| < \frac{n!}{|P|}$, since

$$\lim_{n \to \infty} \frac{f(n) \cdot |\mathcal{P}|}{n!} = 0$$

This is the desired contradiction. □

Theorem 19 *The class of CFP languages is closed under union.*

Proof. Let L_1, L_2 be two CFP languages and $G_1 = (V_1, \Sigma, S_1, \mathcal{P}_1)$, $G_2 = (V_2, \Sigma, S_2, \mathcal{P}_2)$ grammars such that $L(G_1) = L_1$, $L(G_2) = L_2$. Without loss of generality, we can assume $V_1 \cap V_2 = \emptyset$. Let S_0 be a new non-terminal, not contained in $V_1 \cup V_2$, and let

$$\mathcal{P} = \begin{cases} \{S_0 \to \Lambda\} & \text{if } \Lambda \in L_1 \cup L_2 \\ \emptyset & \text{otherwise} \end{cases}$$

Then, the grammar

$$G = (V_1 \cup V_2, \Sigma, S_0, \mathcal{P}_1 \cup \mathcal{P}_2 \cup \{S_0 \to S_1, S_0 \to S_2\} \cup \mathcal{P} \setminus \{S_1 \to \Lambda, S_2 \to \Lambda\})$$

generates the language $L_1 \cup L_2$. □

Theorem 20 *The class of CFP languages is not closed under intersection.*

Proof. Let us define two languages over the alphabet $\Sigma = \{0, 1\}$

$$L_H = \{O \mid O \in \Sigma^{**} \wedge \forall i = 1, \ldots, \text{rows}(O)\ \exists! j : O(i, j) = 1\}$$

$$L_V = \{O \mid O \in \Sigma^{**} \wedge \forall j = 1, \ldots, \text{cols}(O)\ \exists! i : O(i, j) = 1\}$$

The language L_H, resp. L_V consists exactly of all pictures having in each row, resp. column exactly one symbol 1. Both languages can be generated by a CFP grammar. For example, the grammar $G_H = \{\Sigma, \{S, M, R, Z\}, S, \mathcal{P}\}$, where $\mathcal{P}$ consists of the productions

$$S \to \Lambda \qquad S \to M \qquad M \to R \qquad M \to \begin{matrix} R \\ M \end{matrix}$$

$$R \to 1 \qquad R \to \begin{matrix} 1 & Z \end{matrix} \qquad R \to \begin{matrix} Z & 1 \end{matrix} \qquad R \to \begin{matrix} Z & 1 & Z \end{matrix}$$

$$Z \to 0 \qquad Z \to \begin{matrix} 0 & Z \end{matrix}$$

generates L_H. A CFP grammar generating L_V can be constructed analogously.

$L_H \cap L_V$ is the language in Example 11. We proved this language is not a CFP language, thus the class of CFP languages cannot be closed under intersection. □

Theorem 21 *The class of CFP languages is not closed under complement.*

Proof. For a language L, let $\overline{L}$ denote its complement. The intersection of two languages L_1, L_2 can be expressed using union and complement operations as follows

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

thus, with respect to the previous two theorems, CFP languages cannot be closed under complement. □

Theorem 22 *The class of CFP languages is closed under both, row and column, concatenations.*

Proof. We make the proof for the row concatenation (the case of the column concatenation is analogous). Let L_1, L_2 be CFP languages such that $\Lambda \notin L_1 \cup L_2$ and $G_1 = (V_1, \Sigma, S_1, \mathcal{P}_1)$, $G_2 = (V_2, \Sigma, S_2, \mathcal{P}_2)$ CFP grammars such that $L(G_1) = L_1$, $L(G_2) = L_2$. We can assume $V_1 \cap V_2 = \emptyset$. Let S_0 be a new non-terminal, not contained in $V_1 \cup V_2$, Π be the production:

$$S_0 \to \begin{matrix} S_1 \\ S_2 \end{matrix}$$

We put

$$G = (V_1 \cup V_2, \Sigma, S_0, \mathcal{P}_1 \cup \mathcal{P}_2 \cup \{\Pi\})$$

Then, $L(G) = L_1 \ominus L_2$. If $\Lambda \in L_1 \cup L_2$, then $L_1 \ominus L_2$ can be expressed as follows:

- $((L_1 \setminus \{\Lambda\}) \ominus (L_2 \setminus \{\Lambda\})) \cup L_1 \cup L_2$ if $\Lambda \in L_1$ and $\Lambda \in L_2$
- $(L_1 \ominus (L_2 \setminus \{\Lambda\})) \cup L_1$ if $\Lambda \notin L_1$ and $\Lambda \in L_2$
- $((L_1 \setminus \{\Lambda\}) \ominus L_2) \cup L_2$ if $\Lambda \in L_1$ and $\Lambda \notin L_2$

The proof is finished, since we have already proved that CFP languages are closed under union. Also, it should be clear that whenever L is in $L(CFPG)$, $L \setminus \{\Lambda\}$ is in $L(CFPG)$ as well. □

Theorem 23 *The class of CFP languages is closed under projection.*

Proof. Let $G = (V, \Sigma, S_0, \mathcal{P})$ be a CFP grammar, $\pi : \Sigma \to \Gamma$ be a projection. We can construct a CFP grammar $G' = (V, \Gamma, S_0, \mathcal{P}')$, where productions in $\mathcal{P}'$ are modified productions from $\mathcal{P}$ – all occurrences of terminals from Σ in productions in $\mathcal{P}$ are replaced by terminals from Γ, $a \in \Sigma$ being replaced by $\pi(a)$. It should be clear that $\pi(L(G)) = L(G')$.

□

7.8 Emptiness Problem

In section 3.1, we have already mentioned that the emptiness problem is not decidable for FSA's. The idea behind the related proof is as follows. Let T be a one-dimensional deterministic Turing machine, w an input to it, the tape of T be infinite to the right only. A finite computation of T over w can be encoded into a (square) picture – a configuration per a row, where the first field of a row always encodes the first field of the tape. It is easy to see that we can construct a $DFSA$ A which checks whether the first row of an input encodes the initial configuration of T having w as the input, whether the last row encodes a final configuration and whether each row (except the first one) encodes a configuration which is directly reachable from the configuration encoded in the row above. It implies that the question whether T halts on w can be transformed to the question whether A accepts some picture.

CFP grammars does not allow to force local constraints among symbols of two neighboring rows in a picture as A did. However, using a different approach, we show that the emptiness problem is not decidable for CFP grammars as well. Moreover, we will see that this holds even for CFP grammars over one-symbol alphabets.

Theorem 24 *The problem if a given CFP grammar generates some picture is not decidable.*

Proof. Let $T = (\Sigma, \Gamma, Q, q_0, \delta, Q_F)$ be a one-dimensional single-tape deterministic Turing machine, $w \in \Sigma^*$ an input to T. We will consider the tape of T to be infinite to the right only. We show it is possible to construct a CFP grammar

G such that $L(G) \neq \emptyset$ if and only if T accepts w. Since the construction we are going to describe can be performed on a Turing machine, it is evident that the halting problem is transformable to the emptiness problem.

Let the constructed grammar be $G = (V_T, V_N, S, \mathcal{P})$, where $V_T = \{\$\}$. Productions and non-terminals of G will be specified successively. Briefly said, the idea of the construction is to consider an encoding of each possible configuration of T by a positive number and, for a non-terminal C, to find suitable productions such that P is in $L(G, C)$ iff P is a square picture and its length encodes a configuration which T can reach when computing on w. Furthermore, S will generate a picture P iff there is a picture in $L(G, C)$ encoding a configuration preceding an accepting configuration.

For $i > 0$, let $c(i)$ denote the configuration T enters after finishing the i-th step, $c(0)$ be the initial configuration. Denoting $s = |\Gamma|$ and $t = |Q|$, an encoded configuration of T will have the form of a (binary) string over $\{0, 1\}$ starting by 1 followed by concatenations of blocks of the form $1uv$, where $|u| = s$, $|v| = t$. u codes a symbol in Γ. Let $\Gamma = \{\gamma_1, \ldots, \gamma_s\}$. Then, we encode each γ_i by $0^{i-1}10^{s-i}$, i.e. the string of length s consisting of 0's except the i-th position. Similarly, v codes a state in Q. Value 0^t is permitted in this case meaning that no state is encoded. For $\gamma \in \Gamma$, resp. $q \in Q$, we use code(γ), resp. code(q) to denote the string coding γ, resp. q. For our purposes, we make the code of a configuration dependent on the number of a step the configuration has been reached in. To be more precise, let us consider T has just finished the k-th computational step and $a_1 a_2 \ldots a_{k+2}$ are the first $k + 2$ symbols on the tape (starting by the leftmost one). Let the head scan the m-th field (obviously $m \leq k + 2$, even if $|w| = 0$) and T be in a state q. Then, $c(k)$ is encoded as a concatenation of $k + 2$ blocks preceded by the symbol 1:

$$1B_1 B_2 \ldots B_{k+2}$$

Each B_i is of the form $1u_i v_i$, where $u_i = \text{code}(a_i)$ and if $i = m$, then $v_i = \text{code}(q)$, otherwise $v_i = 0^t$.

Let Code(k) denote the code of $c(k)$. We can observe that each value $v = \text{Code}(k)$ can be decomposed into $c(k)$ and k, since

$$k = (\lfloor \log_2 v \rfloor)/(1 + s + t) - 2$$

and $c(k)$ is determined by contents of blocks corresponding to the representation of v in binary.

Now, our goal will be to give a set of productions generating the language $L(G, C)$ such that P is in $L(G, C)$ if and only if P is a square picture and $\text{cols}(P) = \text{Code}(k)$ for some $k \in \mathbb{N}$. To simplify notations, let $p(n)$ denote the square picture over V_T of size Code(n) and bin(P) be the string over $\{0, 1\}$ expressing the size of P written in binary. Moreover, for $w = a_1 \ldots a_n$, where $a_i \in \{0, 1\}$, let $\text{num}(w) = \sum_{i=1}^{n} a_i \cdot 2^{n-i}$, i.e. num($w$) is the number represented by a binary string w.

The first production we add to $\mathcal{P}$ is

$$C \to p(0)$$

The meaning should be clear – C generates the picture which encodes the initial configuration that is reachable at the beginning. Next, we would like to define

productions related to generating of $p(k+1)$ by the assumption that $p(k)$ can be generated. To do it, we will need some auxiliary sets of pictures. We start by the following group of productions:

$$H \to \$ \qquad H \to \begin{array}{cc} \$ & H \end{array} \qquad V \to \$ \qquad V \to \begin{array}{c} \$ \\ V \end{array}$$

$$A \to V \qquad A \to \begin{array}{cc} V & A \end{array} \qquad Q \to \$ \qquad Q \to \begin{array}{cc} Q & V \\ H & \$ \end{array}$$

They are similar to the productions given in Example 7. H generates all non-empty rows over $\{\$\}$, V columns, Q square pictures and A all pictures over $\{\$\}$ except the empty one.

Next, we add three non-terminals D, E, F and productions, such that P is generated by D, resp. E, resp. F iff P is a square picture and $\mathrm{bin}(P)$ is of the form

D) $1B_1B_2 \dots B_k$, where $k \in \mathbb{N}$, each B_i is a block $1u_i0^t$, where u_i codes a symbol in Γ as we have already described (v_i part always equals 0^t).

E) $1B_1B_2 \dots B_kB'B'$ – the same situation as before except the presence of two occurrences of B', each consisting of 0^{1+s+t}, which are appended after the string from D).

F) $1B_1B_2 \dots B_iB'B'B_{i+1} \dots B_k$ – the pair $B'B'$ is inserted inside or appended at the end, following the leading symbol 1 or any block B_i, $i = 1, \dots, k$.

Note that $L(G, E) \subseteq L(G, F)$. Productions related to D are as listed bellow

$$(1) \quad D \to \$$$

For each $i = 1, \dots, s$, there are productions

$$(2) \quad D \to D_i \qquad D_i \to D_{i,s+t+1} \quad (3)$$

$$(4) \quad D_{i,i+1} \to \begin{array}{ccc} D_{i,i} & D_{i,i} & V \\ D_{i,i} & D_{i,i} & V \\ H & H & \$ \end{array} \qquad D_{i,1} \to \begin{array}{ccc} D & D & V \\ D & D & V \\ H & H & \$ \end{array} \quad (5)$$

$$(6) \quad D_{i,j} \to \begin{array}{cc} D_{i,j-1} & D_{i,j-1} \\ D_{i,j-1} & D_{i,j-1} \end{array} \qquad \text{for each } j \in \{2, \dots, s+t+1\} \setminus \{i+1\}$$

Let us take a closer look at the defined productions for some fixed i. We show that $L(G, D_i)$ contains exactly all square pictures P, where $\mathrm{bin}(P)$ is of the form that was expressed in point D) and, moreover, it ends by the suffix $1\,\mathrm{code}(\gamma_i)0^t$.

Assuming D generates square pictures only, to apply production (5) means to substitute the same picture in $L(G, D)$ for all four occurrences of D on the right-hand side of the production. Different square pictures cannot be concatenated

by this scheme. It implies the generated picture is a square again. Let P_1 be some picture of size n in $L(G, D)$ and let P_2 be the picture generated by production (5) when P_1 is substituted to D. Then, P_2 is of size $2n + 1$. It means $\text{bin}(P_2) = \text{bin}(P_1)1$ (the symbol 1 is appended). On the other hand, let us consider a production of type (6) for some possible j, P_1 be a square picture in $L(G, D_{i,j-1})$. If we apply the considered production on P_1 to generate P_2, then P_2 is the square picture of size $2 \cdot \text{cols}(P_1)$. It implies $\text{bin}(P_2) = \text{bin}(P_1)0$. We can generalize these two observations on all productions of types (1)–(6) and summarize them as follows:

- Every picture generated by D, D_i or $D_{i,j}$ is a square picture – production (1) generates the square picture of size 1, all productions (2)–(6) preserve squareness of pictures if a square picture is substituted for D_i or $D_{i,j}$ on right-hand sides.

- Let P' be a picture in $L(G, D)$. Then, the square picture P for which $\text{bin}(P) = \text{bin}(P')1\,\text{code}(\gamma_i)0^t$ is in $L(G, D_i)$. On the other hand, each picture in $L(G, D_i)$ is of a size which fulfils the given expression for some $P' \in L(G, D)$.

  It can be proved as follows. Let $1\,\text{code}(\gamma_i)0^t = b_1 \dots b_{s+t+1}$, where $b_j \in \{0, 1\}$. It means $b_1 = b_{i+1} = 1$, while the other b_j's are equal to 0. Next, for $j = 1, \dots, s + t + 1$, let P_j be the square picture for which $\text{bin}(P_j) = \text{bin}(P')b_1 \dots b_j$. By induction on j, we can easily show that $P_j \in L(G, D_{i,j})$. P_1 is generated by production (5) when P' is substituted for D's on the right-hand side. Next, for $j > 1$, each P_j is generated by production (4), resp. (6), substituting P_{j-1} in proper places in the right-hand side. Since $P = P_{s+t+1}$, production (3) gives the desired result. Moreover, it should be clear that whenever a picture O in $L(G, D_i)$ is generated, the sequence of productions given in the description above is always forced to be applied. It implies the equality $\text{bin}(O) = \text{bin}(O')1\,\text{code}(\gamma_i)0^t$ for some $O' \in L(G, D)$.

  Using the observation inductively, we can see that D generates exactly all the required pictures.

E should generate a square picture P if and only if $\text{bin}(P) = \text{bin}(P')0^{s+t+1}0^{s+t+1}$, where $P' \in L(G, D)$. Speaking in words of the previous considerations, 0 is required to be appended $2 \cdot (s + t + 1)$ times. The following productions perform this operation:

$$E \to E_{s+t+1}$$

$$E_i \to \begin{matrix} E_{i-1} & E_{i-1} \\ E_{i-1} & E_{i-1} \end{matrix} \qquad \text{for each } i = 2, 3, \dots, 2 \cdot (s + t + 1)$$

$$E_1 \to \begin{matrix} D & D \\ D & D \end{matrix}$$

$L(G, F)$ can be generated in the same way as $L(G, D)$. The only difference is that the process must be applied on some picture in $L(G, E)$. The productions are

$$F \to E$$

for each $i = 1, \ldots, s$

$$F \to F_i \qquad F_i \to F_{i,s+t+1}$$

$$F_{i,i+1} \to \begin{matrix} F_{i,i} & F_{i,i} & V \\ F_{i,i} & F_{i,i} & V \\ H & H & \$ \end{matrix} \qquad F_{i,1} \to \begin{matrix} F & F & V \\ F & F & V \\ H & H & \$ \end{matrix}$$

$$F_{i,j} \to \begin{matrix} F_{i,j-1} & F_{i,j-1} \\ F_{i,j-1} & F_{i,j-1} \end{matrix} \qquad \text{for each } j \in \{2, \ldots, s+t+1\} \setminus \{i+1\}$$

Let us consider configurations $c(k)$, $c(k+1)$ for some $k \in \mathbb{N}$. Moreover, let the computational step of T corresponding to the change of configuration from $c(k)$ to $c(k+1)$ be driven by an instruction $I = (q, a) \to (q', a', d)$ of the meaning: T in state q scanning symbol a rewrites a by a', enters state q' and moves the head left if $d = L$, right if $d = R$ or does not move it if $d = N$. We will investigate the difference between Code$(k+1)$ and Code(k). Let $u_1 = \text{bin}(p(k))$ and $u_2 = \text{bin}(p(k+1))$, in $c(k)$, let the head be placed on the m_1-th field of the tape, m_2 be $m_1 - 1$ if $d = L$ and $m_1 + 1$ if $d = R$ or $d = N$. We denote $m = \min(m_1, m_2)$. If u_1 is of the form $1B_1B_2 \ldots B_{k+2}$, then u_2 can be written as

$$1B_1 \ldots B_{m-1}B'_mB'_{m+1}B_{m+2} \ldots B_{k+3}$$

Comparing to u_1, u_2 contains one additional block B_{k+3} storing $1\,\text{code}(\#)0^t$, and blocks B'_m, B'_{m+1} reflecting the changes related to performing the instruction I. Let us define the following values:

- $x_1 = \text{num}(B_mB_{m+1})$, $x_2 = \text{num}(B'_mB'_{m+1})$
- $y = (s+t+1) \cdot (k+1-m)$
- $v = \text{num}(u_1) - x_1 \cdot 2^y$
- $c_1 = s+t+1$, $c_2 = \text{num}(1\,\text{code}(\#)0^t)$

Note that y equals the number of bits that follow after the last bit of B_{m+1} in u_1 and that v equals $\text{num}(u'_1)$, where u'_1 is obtained from u_1 by replacing each bit 1 in B_mB_{m+1} by 0. We can express $\text{num}(u_2)$ to be

$$(v + x_2 \cdot 2^y) \cdot 2^{c_1} + c_2$$

Then, the difference between $\text{num}(u_2)$ and $\text{num}(u_1)$ is as follows

$$\Delta(u_1, u_2) = \text{num}(u_2) - \text{num}(u_1) = (v + x_2 \cdot 2^y) \cdot 2^{c_1} + c_2 - v - x_1 \cdot 2^y =$$

$$= (2^{c_1} - 1) \cdot v + (2^{c_1} \cdot x_2 - x_1) \cdot 2^y + c_2$$

Based on these considerations we will add the following productions to $\mathcal{P}$.

$$(7) \quad C \to \begin{matrix} C & J \\ A & Q \end{matrix} \qquad S \to \begin{matrix} C & J_f \\ A & Q \end{matrix} \quad (8)$$

$$(9)\quad J \to \begin{array}{ccccccc} \overbrace{F & \cdots & F}^{2^{c_1}-1} & A & \overbrace{V & \cdots & V}^{c_2} \\ A & \cdots & A & X & V & \cdots & V \end{array}$$

$$(10)\quad J_f \to \begin{array}{ccccccc} \overbrace{F & \cdots & F}^{2^{c_1}-1} & A & \overbrace{V & \cdots & V}^{c_2} \\ A & \cdots & A & X_f & V & \cdots & V \end{array}$$

$$(11)\quad Q_2 \to \$ \qquad\qquad Q_2 \to \left.\begin{array}{ccc} \overbrace{Q_2 & \cdots & Q_2}^{2^{c_1}} \\ \vdots & \ddots & \vdots \\ Q_2 & \cdots & Q_2 \end{array}\right\} 2^{c_1} \qquad (12)$$

For each instruction $I = (a, q) \to (a', q', d)$, where $q' \notin Q_F$, we add one production of type (13) (x_1 and x_2 are numbers related to I as it has been already described above).

$$X \to \left.\begin{array}{ccc} \overbrace{Q_2 & \cdots & Q_2}^{x_2 \cdot 2^{c_1} - x_1} \\ \vdots & \ddots & \vdots \\ Q_2 & \cdots & Q_2 \end{array}\right\} x_1 \quad (13)$$

Analogously, for each instruction $I = (a, q) \to (a', q', d)$, where $q' \in Q_F$, a production of type (14) is added.

$$X_f \to \left.\begin{array}{ccc} \overbrace{Q_2 & \cdots & Q_2}^{x_2 \cdot 2^{c_1} - x_1} \\ \vdots & \ddots & \vdots \\ Q_2 & \cdots & Q_2 \end{array}\right\} x_1 \quad (14)$$

We will prove that C generates exactly all pictures $p(k)$, where $c(k)$ is not an accepting configuration, while S generates $p(k)$ if and only if $c(k)$ is accepting.

- Production (11) generates the square picture of size 1. If a square picture of size k is substituted in (12) for Q_2, the result is the square picture of size $k \cdot 2^{c_1}$. By induction, Q_2 generates exactly all square pictures of size $2^{c_1 \cdot y}$, where $y \in \mathbb{N}$.

- Considering the previous point, X (resp. X_f) generates pictures of size $x_1 \cdot 2^{c_1 \cdot y} \times (x_2 \cdot 2^{c_1} - x_1) \cdot 2^{c_1 \cdot y}$, where $y \in \mathbb{N}$. Note that $x_2 \cdot 2^{c_1} - x_1$ is always greater than 0, since $x_2 \geq 1$ and $2^{c_1} > x_1$.

- J generates pictures of size $v + y_1 \times (2^{c_1} - 1) \cdot v + y_2 + c_2$, where v is the size of a picture in $L(G, F)$ and $y_1 \times y_2$ the size of a picture in $L(G, X)$. The result for J_f is similar, we need only to replace X by X_f in the previous reasoning.

Now, let us consider production (7). If the production is used to generate a picture, it must be applied on pictures $P_1 \in L(G,C)$ and $P_2 \in L(G,J)$ such that $\text{rows}(P_1) = \text{rows}(P_2)$. Let us assume $P_1 = p(k)$. We have already derived that $\text{rows}(P_2)$ is of the form $v + y_1$, where $y_1 = x_1 \cdot 2^{c_1 \cdot y}$. To find v, x_1 and y satisfying the equality, we will compare binary strings corresponding to sizes. We denote $u_1 = \text{bin}(\text{rows}(P_1))$, $\text{bin}(v)$ is of the form $u = 1B_1 \dots B_{m-1}B'_mB'_{m+1}B_{m+2} \dots B_{l_2}$, $u_1 = 1A_1 \dots A_{l_1}$. After adding $x_1 \cdot 2^{c_1 \cdot y}$ to v, the following conditions must be fulfilled:

- Both symbols at positions corresponding to the first symbols of B'_m and B'_{m+1} must be changed from 0 to 1 (otherwise $v + y_1$ does not encode a configuration). It forces y to be $l_2 - m$ (since blocks are of length at least 3, the case $y = l_2 - m - 1$ cannot change the bit in B'_m).
- The addition does not change length of u, thus l_1 must be equal to l_2. It implies $\text{bin}(x_1) = A_mA_{m+1}$ and $B_i = A_i$ for all $i = 1, \dots, m-2, m+1, \dots, l_1$. Productions related to F guaranteer that the required value v always exists. A suitable x_1 exists only if there is a proper instruction I corresponding to the current state and scanned symbol encoded in A_m or A_{m-1}. Since T is deterministic, there is at most one such an instruction.

So, for given $P_1 \in L(G,C)$, values y and v are determined uniquely, also x_1 if exists. Let us return back to applying production (7). Q must be substituted by the square picture of size $\text{cols}(P_2)$, A is forced to be the picture of size $\text{cols}(P_2) \times \text{cols}(P_1)$. The generated picture is the square picture of size

$$\begin{aligned}&\text{cols}(P_1) + \text{cols}(P_2) = v + x_1 \cdot 2^{c_1 y} + (2^{c_1} - 1) \cdot v + (x_2 \cdot 2^{c_1} - x_1) \cdot 2^{c_1 y} + c_2 = \\ &= 2^{c_1} \cdot v + x_2 \cdot 2^{c_1 y} \cdot 2^{c_1} + c_2 = 2^{c_1} \cdot (v + x_2 \cdot 2^{c_1 y}) + c_2 = \text{Code}(k+1)\end{aligned}$$

By induction on k, we can easily show that $L(G,C)$ contains exactly all the desired pictures. In addition, a picture in $L(G)$ can be generated if and only if there is a configuration $c(k)$ preceding an accepting configuration $c(k+1)$.

□

7.9 Comparison to Other Generalizations of Context-free Languages

In this section we examine some alternative classes of picture languages that have a relation to context-freeness. Namely, we focus on two classes that are mentioned in [2] as possible candidates for the second level of the generalized Chomsky hierarchy.

Languages of the first class are generated by a special type of so called *matrix grammars*. These grammars were proposed by G. Siromoney, R. Siromoney and K. Krithivasan. They consist of two sets of productions – vertical and horizontal respectively. Horizontal productions are used to produce a row of symbols that serve as the initial non-terminals for generating columns using vertical productions. Details on the matrix grammars can be found in [21].

The languages of the second class are defined via systems consisting of two one-dimensional context-free languages (L_1, L_2) and a projection. A picture P

belongs to the language defined by such a system iff it can be obtained as the product of the projection applied on a picture P', where each column, resp. row of P' is a string in L_1, resp. L_2. This class is given in [2] as a suggestion only (without any closer characterization). The proposal is motivated by results on tiling systems (that are an equivalent to OTA's). The class is an extension of $L(OTA)$.

We discuss properties of the classes with respect to their suitability to be promoted to a generalization of context-free languages. We also compare them to $L(CFPG)$.

Definition 20 *A* matrix grammar *is a tuple* $G = (S_0, \mathcal{S}, \Sigma, V_N, V_T, \mathcal{P}_1, \mathcal{P}_2)$, *where*

- $\mathcal{S} = \{\sigma_i\}_{i=1}^n$ *is a finite, non-empty sequence of mutually different elements.*
- *Let* Γ *be an alphabet consisting exactly of all elements in* $\mathcal{S}$. $G(0) = (\Sigma, \Gamma, S_0, \mathcal{P}_1)$ *is a one-dimensional grammar.*
- *For all* $i = 1, \ldots, n$, $G(i) = (V_N, V_T, \sigma_i, \mathcal{P}_2)$ *is a one-dimensional grammar as well.*

Matrix grammars are classified by types of grammars $G(0), G(1), \ldots, G(n)$ into several groups. Since we are interested in context-free grammars, we will consider the case when $G(0)$ and also all $G(i)$'s are context-free. Let $MCFG$ abbreviate a matrix grammar of the described type.

Let P be a picture over V_T, $k = \mathrm{cols}(P)$ and let $P(i)$ denote the i-th column of P. We define a function $\mathrm{ord} : \Gamma \to \mathbb{N}$ as follows

$$\forall i \in \{1, \ldots, n\} \;\; \mathrm{ord}(\sigma_i) = i$$

P is generated by G iff $G(0)$ generates a string $w = x_1 x_2 \ldots x_k$ (where each x_i is in Γ) such that, for each $i = 1, \ldots, k$, $P(i) \in L(G(\mathrm{ord}(x_i)))$. We should remark that, in the literature, the procedure of generating is usually defined in terms of parallel processing and it has to fulfill some restrictions related to the usage of productions. However, for our purposes, it is sufficient to consider the given description, since we are interested in the generated class, which coincides with the class determined by the original definition.

Let $L(MCFG)$ denote the class of all languages generated by $MCFG$'s.

Proposition 18 *$L(MCFG)$ is a proper subset of $L(CFPG)$.*

Proof. It is evident that a $MCFG$ is a special case of a CFP grammar, since right-hand sides of all productions are formed of one-row or one-column matrixes. Furthermore, since each one-dimensional context-free grammar can be replaced by an equivalent grammar in the Chomsky normal form, it should be clear that $MCFG$'s can be simulated even by $CFPG2$'s for which we have already proved that their generative power is less than the generative power of CFP grammars (Theorem 12). □

Definition 21 *A* two-dimensional context-free system *is a tuple* $S = (\Sigma, \Gamma, G_R, G_C, \pi)$, *where*

- Σ *and* Γ *are alphabets.*
- G_R *and* G_C *are one-dimensional context-free grammars over* Γ
- π *is a projection from* Γ *to* Σ*.*

We abbreviate a two-dimensional context-free system by $2CFS$ or $2CF$ system.

Definition 22 *The language generated by a* $2CF$ *system* $S = (\Sigma, \Gamma, G_R, G_C, \pi)$ *is the set* $L(S)$ *containing exactly each picture* P *over* Σ *that can be written as* $P = \pi(P')$*, where* P' *is a picture over* Γ *such that each row, resp. column of* P' *taken as a string is in* $L(G_R)$*, resp.* $L(G_C)$*.*

$L(2CFS)$ denotes the class of all languages that can be generated by a $2CF$ system.

Proposition 19 $L(2CFS)$ *is not a subset of* $L(CFPG)$*.*

Proof. This relation is implied by the fact that $L(OTA)$ is a subset of $L(2CFS)$ [2]. However, to give an example of a $2CFS$, we will prove it directly.

Let us consider the language L in Example 11, i.e. the language over $\Sigma = \{0, 1\}$ consisting of pictures that contain exactly one symbol 1 in each row and column. This language can be generated by the $2CF$ system $S = (\Sigma, \Sigma, G, G, \pi)$, where $\pi : \Sigma \to \Sigma$ is the identity and $G = (\{S_0, Z\}, \Sigma, S_0, \mathcal{P})$, where the productions in $\mathcal{P}$ are:

$$S_0 \to J \qquad J \to 0J \qquad J \to 1Z \qquad Z \to 0Z \qquad Z \to \lambda$$

As we can see, G is even a regular grammar.

On the other hand, we have already proved that $L \notin L(CFPG)$ (Lemma 14). □

The following results regarding NP-completeness are also consequences of the inclusion $L(OTA) \subseteq L(2CFS)$. To demonstrate capabilities of $2CFS$'s, we again prove them directly.

Proposition 20 *To decide whether a* $2CF$ *system* S *generates a picture* P *is an* NP*-complete problem.*

Proof. For a one-dimensional Turing machine $T = (Q, \Sigma_1, \Gamma_1, q_0, \delta, Q_F)$ computing in polynomial time p and an input $w = x_1 \dots x_m$ (where each $x_i \in \Sigma_1$), we show how to construct a $2CF$ system $S = (\Sigma_2, \Gamma_2, G_R, G_C, \pi)$ and a picture P such that $P \in L(S) \Leftrightarrow T$ accepts w. Since we deal with NP-completeness of one dimensional languages, we assume P to be encoded into a string in some way.

The main idea of the construction is based on encoding a computation of T into a picture, where each row encodes one configuration, starting by the initial configuration in the first row and ending by a final configuration in the last row. S will generate all pictures encoding accepting computations.

Without loss of generality, we assume

- T never moves to the left from the filed scanned in the initial configuration, i.e. from the field corresponding to the leftmost symbol of w if $w \neq \lambda$.
- During the first step, T always changes its state from q_0 to q_1, it does not rewrite the scanned symbol and does not move the head. It means, there is exactly one instruction applicable on the initial configuration and this instruction is of the given form.
- T has exactly one final state. Let this state be q_f.
- If T reaches q_f, then the final configuration is always $\#q_f$. Before T accepts, all symbols different to # are erased in the tape and the head is moved to the field scanned in the initial configuration.
- T always performs at least $|w|$ steps (this condition is implied by the previous point).

Let $\mathcal{I}$ be the set of all instructions of T. For a convenience, we assume $\mathcal{I}$ contains also a fictive instruction I_f that is applicable on each final configuration and does not perform any changes (no rewriting, no head movement). We define

$$\Sigma_2 = \Gamma_1 \cup (\Gamma_1 \times \mathcal{I}) \cup (\Gamma_1 \times Q \times \mathcal{I})$$

and $\Gamma_2 = \Sigma_2 \times \{0, 1\}$. For any $(b, d) \in \Gamma_2$, let

$$\pi((b, d)) = \begin{cases} b & \text{if } d = 1 \\ \# & \text{if } d = 0 \end{cases}$$

For all $(b, d) \in \Gamma_2$, let $\phi((b, d)) = b$ and for all $v = y_1 \dots y_k$, where each $y_i \in \Gamma_2$, let $\phi(v) = \phi(y_1) \dots \phi(y_k)$. To achieve a better readability in the following text, for each $(b, d) \in \Gamma_2$, we will omit the d-part and write b only. This step can be justified by the fact that whenever w will be in $L(G_R)$, resp. $L(G_C)$, the language will be required to contain also each w' for which $\phi(w') = \phi(w)$.

G_R is designed to generate strings encoding a configuration of T and an instruction T is going to perform. Let $v = a_1 a_2 \dots a_n \in \Gamma_1^*$ ($a_i \in \Gamma_1$), $q \in Q$, k be a positive integer not greater than n and $I \in \mathcal{I}$ be an instruction applicable on the configuration $a_1 a_2 \dots a_k q a_{k+1} \dots a_n$. Moreover, if I moves the head left, resp. right, we assume $k > 1$, resp. $k < n$. Depending on the movement, G_R generates

- $a_1 a_2 \dots a_{k-2}(a_{k-1}, I)(a_k, q, I)a_{k+1} \dots a_n$ (movement left)
- $a_1 a_2 \dots a_{k-1}(a_k, q, I)a_{k+1} \dots a_n$ (no movement)
- $a_1 a_2 \dots a_{k-1}(a_k, q, I)(a_{k+1}, I)a_{k+1} \dots a_n$ (movement right)

A triple of the form (a, q, I) determines the current position of the head, state and an instruction to be applied. A tuple of the form (a, I) determines a new position the head will be moved to after applying I.

G_C generates each string $c_1 \dots c_n$, where for every $i = 1, \dots, n-1$, c_i and c_{i+1} fulfill

- If $c_i = (a, q, I)$, where I rewrites a by a' and changes q to q', then

- $c_{i+1} = (a', q', I')$ if I does not move the head (I' is an arbitrary instruction, not necessary applicable, since the applicability is being checked by the grammar G_R). This ensures that, after performing I, the head of T remains at the same position.
- $c_{i+1} = (a', I')$ or $c_{i+1} = a'$ if I moves the head. The first case reflects the situation when the head is moved back in the next step applying I', while in the second case, a movement back does not occur.

- If $c_i = (a, I)$, then $c_{i+1} = (a, q', I')$. c_i indicates that the head was moved to this position by I, thus c_{i+1} must be a triple of the given form.

- If $c_i = a$, then $c_{i+1} = a$ or $c_{i+1} = (a, I')$.

Informally, G_C is designed to check in columns that instructions encoded together with configurations are really performed and changes made by them reflected in following configurations.

All presented requirements on strings in $L(G_R)$ and $L(G_C)$ can be checked by a finite-state automaton, thus G_R and G_C can be even regular grammars.

P is a square picture of size $p(|w|)$ given by the following scheme:

(x_1, q_0, I_0)	x_2	$\ldots$	x_n	#	$\ldots$	#
#						#
$\vdots$						$\vdots$
$(\#, q_f, I_f)$	#		$\ldots$			#

I_0, resp. I_f is the only instruction applicable on the state q_0, resp. q_f. The first row of P encodes the initial configuration of T, the last row the final configuration. Remaining rows are formed of #'s. Let P_1 be a picture over Γ_2 such that $P = \pi(P_1)$ and all rows resp. columns of P_1 are in $L(G_R)$, resp. $L(G_C)$. It should be clear that rows of P_1 encode an accepting computation of T. On the other hand, if there is an accepting computation, a picture proving $P \in L(S)$ can be constructed.

So far, we have proved the NP-hardness of the problem. To finish the proof it is sufficient to realize that if there is a picture P_1 proving $P \in L(S)$, it can be non-deterministically guessed and verified in polynomial time, and vice versa.

□

Proposition 21 $L(2CFS) \cap NP_{2d} \neq \emptyset$

Proof. Let L_1 be a one-dimensional NP-complete language over Σ, T be a one-dimensional Turing machine recognizing L_1 in polynomial time p. Since the construction of S in the previous proof was nearly independent on the input string w, we can use the same approach to construct a $2CF$ system for T (assuming T fulfills all requirements listed in the proof), but this time with additional minor modifications. Strings in $L(G_R)$, resp. $L(G_C)$ are required to satisfy:

- In a row, if q_0 appears in the represented configuration, then the configuration must be initial.

- In a column, if a state is encoded in the first, resp. last field, it must be q_0, resp. q_f.
- In a column, the first and last field of the column must store a symbol of the form $(b, 1)$, where $b \in \Sigma_2$, while the other fields symbols of the form $(b, 0)$.

These three additional requirements ensure that S generates exactly each picture P', where the first row encodes an initial configuration, the last row an accepting configuration and the other rows are strings of #'s. Moreover, T accepts the encoded input string in at most rows(P') steps using at most cols(P') fields of space.

$L_2 = L(S)$ is NP_{2d}-complete, since there is a polynomial transformation of L_1 to L_2. Each $w \in \Sigma^*$ is transformed to the picture P as it was described in the proof of Proposition 20. □

We have already proved that languages in $L(CFPG)$ can be recognized in polynomial time (Theorem 15). We can conclude that $2CF$ systems are too strong to be considered as a tool defining the generalization of context-free languages.

Chapter 8

Conclusions

In the thesis we have confirmed the fact that there are many differences in results between the theory of two-dimensional languages and the classical one. They are caused by a more complex topology of pictures, which is, comparing to strings, more contextual. Some of the most important differences we have found are summarized in the following paragraphs.

- A two-dimensional finite-state automaton is a natural candidate for the recognizing device defining the ground level class of picture languages, however, $L(FSA)$ has some properties that differ to those known from the one-dimensional case. Namely, we have shown that $L(FSA)$ is not closed under projection (Theorem 2). On the other hand, two-dimensional on-line tessellation automata seems to be too strong, since they are able to recognize some NP_{2d}-complete problems. We have shown that for every bounded one-dimensional cellular automaton, an OTA simulating it can be constructed (Theorems 9 and 10).

- Two-dimensional grammars with productions in context-free form are a straightforward generalization of context-free grammars, but the properties of the class $L(CFPG)$ prevent to promote $L(CFPG)$ to a generalization of context-free languages. We have seen that the class is incomparable with $L(FSA)$ (Theorem 13), there is no normal form of productions, since the generative power depends on size of production's right-hand sides (Theorem 16), etc.

- We have shown that, comparing to a one-dimensional tape, a two-dimensional tape is an advantage, since some computations can be done substantially faster there. We have demonstrated this fact on palindromes (section 4.2). Two-dimensional Turing machine can decide in time $O(n^2/\log n)$ whether an input string is a palindrome. This cannot be done in better than quadratic time when a one-dimensional tape is used. We have also proved that our algorithm is optimal (section 4.4).

Throughout the thesis we have investigated relationships among the classes of picture languages defined by the models. In Figure 8, we can see a scheme where our results are summarized together with already known facts.

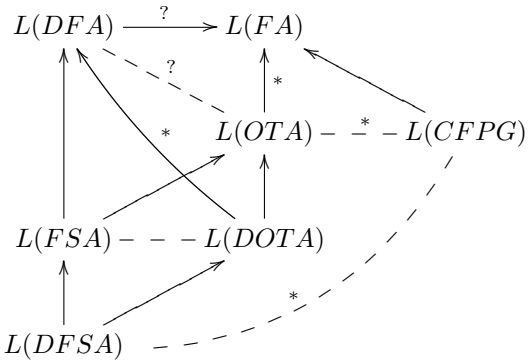


Figure 8.1: Relations among the classes we have studied. An arrow leading from class C_1 to class C_2 denotes that C_1 is a proper subset of C_2 while dashed line connecting two classes indicates that the classes are incomparable. Non-labelled relations are already known results. Our results are labelled by $*$ and ?. The question mark indicates that the related result has been proved by the assumption $P \neq NP$.

In the remaining part of this chapter we give a list of problems that have not been solved and that, in our opinion, are important and deserve an attention. They are related mostly to CFP grammars.

Recognizing device for L(CFPG)

We have not succeeded in finding a model of device recognizing exactly $L(CFPG)$. We have shown that two-dimensional forgetting automata can simulate CFP grammars (Theorem 14), however, they are able to recognize also some languages outside the class $L(CFPG)$. It is a question if some suitable modifications of the model can be done to obtain the required device. Of course, we can look for a model characterizing the class based on different principles than forgetting as well.

One of the approaches can be to study devices equipped with a pushdown store. Even if we consider a two-dimensional finite-state automaton having this storage available we get a device of recognitive power surely greater comparing to FSA's. It seems, this sort of two-dimensional automata have not been studied much yet.

Emptiness problem in case of CFPG2

In section 7.8, we have proved that the emptiness problem is not decidable for CFP grammars. In the presented construction, it was essential that we worked with productions having right-hand sides of size 2×2. The arising question is if the emptiness problem is still undecidable if we restrict $CFPG$ to $CFPG2$.

Language of pictures consisting of one component

To show that a language L is not in $L(CFPG)$, we used the technique based on classifying pictures in L with respect to by which productions they can be generated in the first step (see, e.g., the proof of Theorem 10). This technique was sufficient for our purposes, however, in [9], there is an open problem stated

that we have not solved applying the technique. Let us consider the language described in Example 12.

Example 12 Let $\Sigma = \{a, b\}$, P be a picture over Σ and $g(P)$ a non-oriented graph assigned to P, where nodes correspond to fields of P that store b and an edge connects two nodes if and only if the related fields in P are neighbors (we consider each field to have four neighbors maximally – the left, right, top and bottom neighbor). We say that P contains one component of b's iff $g(P)$ is formed of one component. Let L_c be the language consisting exactly of all pictures over Σ that contain one component of b's.

There is a hypothesis that L_c cannot be generated by a CFP grammar. It is difficult to apply our technique to confirm the hypothesis, since, roughly said, there are many variants of pictures in L_c. The problem remains still open although we have presented the technique, which was not considered in the mentioned resource.

Possible extension of CFPG

We have proved that $L(CFGP)$ and $L(FSA)$ are incomparable (Theorem 13). If we would like to have grammars that cover languages recognizable by two-dimensional finite-state automata, we can consider the following extension:

Definition 23 An extended two-dimensional grammar with productions in context-free form *is a tuple* $(V_N, \mathcal{V}_S, S_0, \mathcal{R}, \mathcal{P})$*, where*

- V_N *is a finite set of non-terminals*
- $S_0 \in V_N$ *is the initial non-terminal*
- $\mathcal{V}_S = \{\sigma_i\}_{i=1}^n$ *is a finite, non-empty sequence of different starting symbols*
- $\mathcal{R} = \{L_i\}_{i=1}^n$ *is a finite sequence of length* $|\mathcal{V}_S|$ *of languages in* $L(FSA)$
- *Let* V_S *be the set consisting exactly of all elements in* $\mathcal{V}_S$*. Then,* $\mathcal{P}$ *is a finite set of productions of the form* $N \to W$*, where* $N \in V_N$ *and* $W \in (V_N \cup V_S)^{**} \setminus \{\Lambda\}$*. In addition,* $\mathcal{P}$ *can contain* $S_0 \to \Lambda$*. In this case, no production in* $\mathcal{P}$ *contains* S_0 *as a part of its right-hand side.*

Let $ECFPG$ stand for an extended two-dimensional grammar with productions in context-free form. Comparing to $CFPG$, $ECFPG$ does not contain terminals, but rather starting symbols, where each σ_i can be substituted by any non-empty picture in L_i. More precisely, for $V \in V_N \cup V_S$, we define $L(G, V)$, the language generated by V in G, as follows.

1. If $S_0 \to \Lambda$ is in $\mathcal{P}$ then $\Lambda \in L(G, S_0)$.
2. For each $\sigma_i \in V_S$, $L(G, \sigma_i) = L_i$.
3. Let $N \to [A_{ij}]_{m,n}$ be a production in $\mathcal{P}$, different to $S_0 \to \Lambda$, and P_{ij} $(i = 1, \ldots, n;\ j = 1, \ldots, m)$ non-empty pictures such that $P_{ij} \in L(G, A_{ij})$. Then, if $\bigoplus [P_{ij}]_{m,n}$ is defined, it is in $L(G, N)$.

The language generated by G is $L(G) = L(G, S_0)$. Having defined the extended variant of CFP grammars, we can make two observations.

1. $L(CFPG) \subseteq L(ECFPG)$ – For any CFP grammar $G = (V_N, V_T, S_0, \mathcal{P})$, where $V_T = \{a_1, \ldots, a_n\}$, we can construct $ECFP$ grammar $G' = (V_N, \{\sigma_i\}_{i=1}^n, S_0, \mathcal{R}, \mathcal{P})$, where $\mathcal{R} = \{L_i\}_{i=1}^n$ and each $L_i = \{a_i\}$. It should be clear that $L(G') = L(G)$.

2. $L(FSA) \subseteq L(ECFPG)$ – Let L_F be a language in $L(FSA)$. Then $G = (\{S_0\}, \{\sigma\}, S_0, \{L_F\}, \mathcal{P})$, where $\mathcal{P} = \{S_0 \to \sigma\}$ if $\Lambda \notin L_F$, otherwise $\mathcal{P} = \{S_0 \to \sigma, S_0 \to \Lambda\}$, is a $ECFPG$ generating L_F.

Except the observations, it is also clear that if productions of the grammars are restricted to have one row on their right-hand sides only (or possibly Λ), then the grammars generate exactly context-free languages.

Questions regarding properties of the class $L(ECFPG)$ arise here.

Relationship between FA and DFA

We have shown that by the assumption $NP \neq P$, FA's are stronger than DFA's, since FA's can recognize some NP_{2d}-complete problems (Proposition 12). It is a question if a language $L \in P_{2d}$ such that $L \in L(FA)$ and $L \notin L(DFA)$ can be found to separate these two classes. The open status of this problem implies it is also open for the one-dimensional variant of forgetting automata.

Bibliography

[1] P. Foltýn: *Zapomínací automat s rovinným vstupem*, Master thesis, Faculty of Mathematics and Physics, Charles University, Prague, 1994, (in Czech).

[2] D. Giammarresi, A. Restivo: *Two-dimensional languages*, in A. Salomaa and G. Rozenberg, editors, Handbook of formal languages, volume 3, Beyond Words, pp. 215–267, Springer-Verlag, Berlin, 1997.

[3] D. Giammarresi, A. Restivo: *Recognizable picture languages*, Int. J. of Pattern Recognition and Artificial Inteligence 6(2-3), pp. 32–45, 1992.

[4] J. Hopcroft, J. Ullman: *Formal languages and their relation to automata*, Addison-Wesley, 1969.

[5] K. Inoue, A. Nakamura: *Some properties of two-dimensional on-line tessellation acceptors*, Information Sciences, Vol. 13, pp. 95–121, 1977.

[6] K. Inoue, A. Nakamura: *Two-dimensional finite automata and unacceptable functions*, Intern. J. Comput. Math., Sec. A, Vol 7, pp. 207–213, 1979.

[7] P. Jančar, F. Mráz, M. Plátek: *A taxonomy of forgetting automata*, in proceedings of MFCS 1993, LNCS 711, pp. 527–536, Springer, 1993.

[8] P. Jančar, F. Mráz, M. Plátek: *Forgetting automata and context-free languages*, Acta Informatica 33, pp. 409–420, Springer-Verlag, 1996.

[9] P. Jiřička: *Grammars and automata with two-dimensional lists (grids)*, Master thesis, Faculty of Mathematics and Physics, Charles University, Prague, 1997, (in Czech).

[10] P. Jiřička, J. Král: *Deterministic forgetting planar automata*, in proceedings of the 4-th Int. Conference: Developments in Language Theory, Aachen, Germany, 1999.

[11] J. Kari, Ch. Moore: *Rectangles and squares recognized by two-dimensional automata*, In Theory Is Forever, Essays Dedicated to Arto Salomaa on the Occasion of His 70th Birthday. J.Karhumaki, H.Maurer, G.Paun, G.Rozenberg (Eds.), LNCS 3113, pp. 134–144.

[12] K. Lindgren, Ch. Moore, M. Nordahl: *Complexity of two-dimensional patterns*, Journal of Statistical Physics 91(5-6), pp. 909–951, 1998.

[13] O. Matz: *Regular Expressions and Context-free Grammars for Picture Languages*, In 14th Annual Symposium on Theoretical Aspects of Computer Science, volume 1200 of Lecture Notes in Computer Science, pp. 283–294, Springer-Verlag, 1997.

[14] D. Průša, F. Mráz: *Parallel Turing machines on a two-dimensional tape*, In proceedings of the Czech Pattern Recognition Workshop, ČVUT, Peršlák, 2000.

[15] D. Průša: *Recognition of palindromes*, In proceedings of WDS 2000, Charles University, Prague, 2000.

[16] D. Průša: *Two-dimensional Context-free Grammars*, In proceedings of ITAT 2001, pp. 27–40. Zuberec, 2001.

[17] D. Průša: *Two-dimensional on-line tessellation automata: Recognition of NP-complete problems*, In proceedings of ITAT 2003, Sliezsky Dom, 2003.

[18] D. Průša: *Two-dimensional Grammars in Context-free Form: Undecidability of the Emptiness Problem*, In proceedings of MIS 2004, Josefův Důl, Matfyzpress, 2004.

[19] D. Průša: *Paralelní Turingovy stroje nad dvojrozměrnými vstupy*, Master thesis, Faculty of Mathematics and Physics, Charles University, Prague, 1997, (in Czech).

[20] A. Rosenfeld: *Picture languages – formal models of picture recognition*, Academic Press, New York, 1979.

[21] G. Siromoney, R. Siromoney, K. Krithivasan: *Abstract families of matrices and picture languages*, Comput. Graphics and Image Processing 1, pp. 284–307, 1972.

[22] M.I. Schlesinger, V. Hlaváč: *Ten lectures on statistical and syntactic pattern recognition*, Kluwer Academic Publishers, 2002.