Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS

Lukáš Krtek

Learning picture languages using restarting automata

Department of Software and Computer Science Education

Supervisor of the master thesis:	RNDr. František Mráz, CSc.
Study programme:	Computer Science
Specialization:	Theoretical Computer Science

Prague 2014

I would like to thank RNDr. František Mráz, CSc. for his kind and patient guidance, for providing invaluable advice and for all the time he has invested in supervising this thesis.

Název práce: Učení obrázkových jazyků pomocí restartovacích automatů

Autor: Lukáš Krtek

Katedra: Kabinet software a výuky informatiky

Vedoucí diplomové práce: RNDr. František Mráz, CSc., Kabinet software a výuky informatiky

Abstrakt: Ačkoliv existuje mnoho modelů automatů pracujících nad dvojrozměrnými vstupy (obrázky), málokdo se dosud zabýval tématem učení těchto automatů. V této práci představujeme nový model zvaný dvojrozměrný restartovací automat s omezeným kontextem. Náš model pracuje podobně jako dvojrozměrný restartovací dlaždicový automat, avšak ukazuje se, že má stejnou sílu jako dvojrozměrný sgrafito automat. V práci jsme navrhli algoritmus učení těchto automatů z pozitivních a negativních příkladů obrázků. Tento algoritmus je implementován a následně otestován na několika základních obrázkových jazycích.

Klíčová slova: obrázkový jazyk, gramatická inference, restartovací automat

Title: Learning picture languages using restarting automata

Author: Lukáš Krtek

Department: Department of Software and Computer Science Education

Supervisor: RNDr. František Mráz, CSc., Department of Software and Computer Science Education

Abstract: There are many existing models of automata working on two-dimensional inputs (pictures), though very little work has been done on the subject of learning of these automata. In this thesis, we introduce a new model called two-dimensional limited context restarting automaton. Our model works similarly as the two-dimensional restarting tiling automaton, yet we show that it is equally powerful as the two-dimensional sgraffito automaton. We propose an algorithm for learning of such automata from positive and negative samples of pictures. The algorithm is implemented and subsequently tested with several basic picture languages.

Keywords: picture language, grammatical inference, restarting automaton

Contents

Introduction

The first formal model of automaton working on two-dimensional tape was the four-way finite automaton introduced in 1967 ([5]). This model is a direct extension of the one-dimensional finite state automaton, but has some undesirable properties. Because of that, other models were considered for a suitable generalization of the class of regular languages for pictures. The most notable candidate for this extension is the class of recognizable tiling languages. Introduced in 1991, the class was initially defined by so-called tiling systems ([9]). Another recent candidate is the class induced by the two-dimensional sgraffito automaton model ([18]), which is a variant of the two-dimensional Turing machine. Both of these classes, when restricted to one-dimensional inputs, yield the class of regular languages.

Of course, there are many other two-dimensional models of automata inducing many different classes of languages. Of special interest to us is the recently introduced restarting tiling automaton ([17]), which works by cyclic rewriting of an input picture. This mode of operation makes the model an interesting choice for procedural learning of picture languages.

In this thesis, we take the restarting tiling automaton and simplify it in hope of creating a model that is even more suitable for learning. We introduce a new model called the two-dimensional limited context restarting automaton and study its properties, e.g. closure properties and the relation to the sgraffito automaton. With this model, we proceed to fulfill our main goal, that is to propose and implement an algorithm for learning picture languages from positive and negative samples.

Perhaps due to the complex nature of the problem, there are virtually no existing algorithms of this kind. This fact is in stark contrast with the situation in the field of one-dimensional languages, where a multitude of such algorithms exists ([10], [4], [7] etc.).

Our thesis consists of six chapters. In the first chapter, we establish the basic terminology of picture languages and list several existing models of automata. In the second chapter, we define our new model of a restating automaton and its so-called correctness preserving variant. Chapter 3 demonstrates the usage of our model on some specific picture languages. In the fourth chapter, we explore the properties of our model and compare it to some existing models, most notably the sgraffito automaton. In the fifth chapter, we propose an algorithm for simulation of our model and, more importantly, the algorithm for learning from positive and negative samples of pictures. Finally, in the last chapter, we test our implementation of the proposed learning algorithm on a small collection of basic picture languages. The implementation itself is documented in the Appendix.

Chapter 1

Existing Models

In this chapter we will list some relevant existing models of automata working with two-dimensional inputs. But first we need to establish the basic definitions and terminology of picture languages.

1.1 Basic definitions

The notation used for picture languages is almost the same in all the works on the subject, and we will certainly abide by it. When referring to natural numbers, we denote by $\mathbb{N}$ the set $\{1, 2, \dots\}$ and by $\mathbb{N}_0$ the set $\{0, 1, 2, \dots\}$. The rest of this section is taken mostly from [17].

Definition 1. *A* picture *over a finite alphabet* Σ *is a two-dimensional rectangular array of elements from* Σ*. The set of all pictures over* Σ *is denoted by* $\Sigma^{*,*}$*. A* picture language *over* Σ *is a subset of* $\Sigma^{*,*}$*.*

Let P be a picture over Σ. We denote the number of rows and columns of P by $\text{rows}(P)$ and $\text{cols}(P)$, respectively. The pair $(\text{rows}(P), \text{cols}(P))$ is called the *size* of P. The *empty picture* Λ is defined as the only picture of the size $(0, 0)$. The set of all pictures of size (m, n) over Σ is denoted by $\Sigma^{m,n}$. Assuming $1 \leq i \leq \text{rows}(P)$ and $1 \leq j \leq \text{cols}(P)$, $P(i, j)$ (or shortly $P_{i,j}$) identifies the symbol in the i-th row and the j-th column in P.

Two (partial) binary operations are used to concatenate pictures. Let P and Q be pictures over Σ of sizes (k, l) and (m, n), respectively. The *horizontal concatenation* $P \mathbin{⦶} Q$ is defined iff $k = m$, the *vertical concatenation* $P \ominus Q$ is defined iff $l = n$. The corresponding products are depicted below:

$$P \mathbin{⦶} Q = \begin{matrix} P_{1,1} & \cdots & P_{1,l} & Q_{1,1} & \cdots & Q_{1,n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ P_{k,1} & \cdots & P_{k,l} & Q_{m,1} & \cdots & Q_{m,n} \end{matrix} \quad P \ominus Q = \begin{matrix} P_{1,1} & \cdots & P_{1,l} \\ \vdots & \ddots & \vdots \\ P_{k,1} & \cdots & P_{k,l} \\ Q_{1,1} & \cdots & Q_{1,n} \\ \vdots & \ddots & \vdots \\ Q_{m,1} & \cdots & Q_{m,n} \end{matrix}$$

We also define $\Lambda \ominus P = P \ominus \Lambda = \Lambda \mathbin{⦶} P = P \mathbin{⦶} \Lambda = P$ for any picture P.

In addition, we introduce the *clockwise rotation* P^{R}, *vertical mirroring* P^{VM} and *horizontal mirroring* P^{HM}.

$$P^{\mathrm{R}} = \begin{array}{ccc} P_{k,1} & \cdots & P_{1,1} \\ \vdots & \ddots & \vdots \\ P_{k,l} & \cdots & P_{1,l} \end{array} \quad P^{\mathrm{VM}} = \begin{array}{ccc} P_{1,l} & \cdots & P_{1,1} \\ \vdots & \ddots & \vdots \\ P_{k,l} & \cdots & P_{k,1} \end{array}$$

$$P^{\mathrm{HM}} = \begin{array}{ccc} P_{k,1} & \cdots & P_{k,l} \\ \vdots & \ddots & \vdots \\ P_{1,1} & \cdots & P_{1,l} \end{array}$$

Remark Note that the vertical mirroring can be expressed by the horizontal mirroring and rotation (and vice-versa).

$$P^{\mathrm{VM}} = \left(\left(P^{\mathrm{HM}}\right)^{\mathrm{R}}\right)^{\mathrm{R}}$$

Let $\pi : \Sigma \to \Gamma$ be a mapping between two alphabets. The projection by π of $P \in \Sigma^{m,n}$ is the picture $P' \in \Gamma^{m,n}$ such that $P'(i,j) = \pi(P(i,j))$ for all $1 \le i \le m, 1 \le j \le n$, we write $P' = \pi(P)$.

We can extend all introduced operations to languages. Let L, L' be picture languages over Σ and $\pi : \Sigma \to \Gamma$ be a mapping. Then

$$\begin{aligned} L \oplus L' &= \{P \oplus P' \mid P \in L \wedge P' \in L'\}, \\ L \ominus L' &= \{P \ominus P' \mid P \in L \wedge P' \in L'\}, \\ \pi(L) &= \{\pi(P) \mid P \in L\}, \\ L^{\mathrm{R}} &= \{P^{\mathrm{R}} \mid P \in L\}, \\ L^{\mathrm{VM}} &= \{P^{\mathrm{VM}} \mid P \in L\}, \\ L^{\mathrm{HM}} &= \{P^{\mathrm{HM}} \mid P \in L\}. \end{aligned}$$

Let $\mathcal{S} = \{\vdash, \dashv, \top, \bot, \#\}$ be a set of special markers (*sentinels*), Σ be a finite alphabet such that $\Sigma \cap \mathcal{S} = \emptyset$ and $P \in \Sigma^{m,n}$ be a picture over Σ of size (m,n). We define a *boundary picture* of P as a picture $\widehat{P}$ over $\Sigma \cup \mathcal{S}$ of size $(m+2, n+2)$. Its symbols are given by the following scheme.

$$\begin{array}{|c|c|c|} \hline \# & \top \ \top \ \cdots \ \top \ \top & \# \\ \hline \vdash & & \dashv \\ \vdots & P & \vdots \\ \vdash & & \dashv \\ \hline \# & \bot \ \bot \ \cdots \ \bot \ \bot & \# \\ \hline \end{array}$$

Formally

$$\widehat{P}(i,j) = \begin{cases} P(i-1, j-1) & \text{for } 2 \le i \le m+1 \text{ and } 2 \le i \le n+1, \\ \vdash & \text{for } 2 \le i \le m+1 \text{ and } j = 1, \\ \dashv & \text{for } 2 \le i \le m+1 \text{ and } j = m+2, \\ \top & \text{for } i = 1 \text{ and } 2 \le j \le n+1, \\ \bot & \text{for } i = m+2 \text{ and } 2 \le j \le n+1, \\ \# & \text{for } (i,j) \in \{f(1,1), (1, n+2), (m+2, 1), (m+2, n+2)\}. \end{cases}$$

We call a *tile* each picture of the size $(2,2)$. Given a picture P, we denote by $B_{2,2}(P)$ the set of all tiles that are sub-pictures of $\widehat{P}$.

1.2 Tiling systems and the class REC

Introduced in [9], the class of recognizable picture languages (REC) was first defined using tiling systems. This class is usually considered to be a base class of picture languages. Indeed, it is often thought to be the most suitable generalization of one-dimensional regular languages into two dimensions.

To define tiling systems, we need to introduce the notion of local picture languages, which will also be useful later for defining restarting tiling automata.

Both of the following definitions are taken from [8].

Definition 2. *A two-dimensional language $L \subseteq \Gamma^{*,*}$ is* local *if there exists a set Θ of tiles over $\Gamma \cup \mathcal{S}$ such that $L = \{p \in \Gamma^{*,*} \mid B_{2,2}(p) \subseteq \Theta\}$ and we will write $L = L(\Theta)$.*

Definition 3. *A two-dimensional language $L \subseteq \Sigma^{*,*}$ is recognized by a* tiling system $(\Sigma, \Gamma, \Theta, \pi)$ *if $L = \pi(L(\Theta))$. A language is* tiling recognizable *if it is recognized by some tiling system. The family of all tiling recognizable picture languages is denoted by* REC*.*

Recognizing a language in REC can be NP-hard and so it makes sense to define a deterministic subclass (DREC) whose languages are recognizable in polynomial time. The following definition also comes from [8].

Definition 4. *A tiling system $(\Sigma, \Gamma, \Theta, \pi)$ is* tl2br-deterministic (top left to bottom right deterministic) *if for any $\gamma_1, \gamma_2, \gamma_3 \in (\Gamma \cup \mathcal{S})$ and $\sigma \in \Sigma$ there exists at most one tile $\begin{array}{|c|c|}\hline \gamma_1 & \gamma_2 \\ \hline \gamma_3 & \gamma_4 \\ \hline \end{array} \in \Theta$, with $\pi(\gamma_4) = \sigma$.*

Similarly we define d-deterministic tiling systems for any corner-to-corner direction d. A recognizable two-dimensional language L is deterministic*, if it admits a d-deterministic tiling system for some corner-to-corner direction d. We denote by* DREC *the class of* Deterministic Recognizable Two-dimensional Languages*.*

While a tiling system gives us a simple and comprehensive definition of a language in REC, it is not a model of automaton with a defined operation which could be simulated. For that reason, several models of automata were introduced that recognize the same class of languages, like the tiling automaton (introduced in [1]) or the two-dimensional on-line tessellation acceptor (introduced in [11]).

We recount a concise definition of two-dimensional on-line tessellation acceptors (2OTA) from [6].

Definition 5. *A* 2OTA *is a 5-tuple $\mathcal{A} = (\Sigma, Q, I, F, \delta)$, where Σ is the input alphabet, Q is the set of states, $I \subseteq Q, F \subseteq Q$ are the sets of initial and final states, $\delta : Q \times Q \times \Sigma \to 2^Q$ is the transition function.*

A run of $\mathcal{A}$ over a picture $p \in \Sigma^{*,*}$ associates a state to each position of p. At time $t = 0$ a state $q_0 \in I$ is associated to all positions of the first row and column of $\widehat{p}$, then moving diagonally across the array, at time $t = k$, states are simultaneously associated to each position (i, j) of the picture with $i + j - 1 = k$, according to δ, using the states associated with positions $(i - 1, j)$ $(i, j - 1)$ and the input symbol $p(i, j)$. The picture p is recognized by $\mathcal{A}$ if there is a run of $\mathcal{A}$ associating a final state to the position $(\text{rows}(p), \text{cols}(p))$*.*

1.3 Two-dimensional sgraffito automaton

The two-dimensional sgraffito automaton is a variant of the original Turing machine working on two-dimensional inputs. In contrast to a Turing machine, every computation of a sgraffito automaton is finite because it must reduce finite, non-negative integer weights of its tape in every step.

The following formal definitions are taken from [17].

Let $\mathcal{H} = \{\mathrm{R}, \mathrm{L}, \mathrm{D}, \mathrm{U}, \mathrm{Z}\}$ be the set of *head movements*. First four elements on $\mathcal{H}$ denote directions: left, right, up, down. Z stands for zero (none) movement. We define a mapping $\nu : S \to \mathcal{H}$ such that

$$\nu(\vdash) = \mathrm{R}, \nu(\dashv) = \mathrm{L}, \nu(\top) = \mathrm{D}, \nu(\bot) = \mathrm{U} \text{ and } \nu(\#) = \mathrm{Z}.$$

Definition 6. *A (non-deterministic)* two-dimensional bounded Turing machine*, bounded* 2TM *for short, is a tuple $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_0, Q_F)$ where*

- *Σ is an input alphabet, Γ is a working alphabet such that $\Sigma \subseteq \Gamma$ and $\Gamma \cap \mathcal{S} = \emptyset$,*
- *Q is a finite, nonempty set of states,*
- *$q_0 \in Q$ is the initial state,*
- *$Q_F \subseteq Q$ is the set of final states, and*
- *$\delta : (Q \setminus Q_F) \times (\Gamma \cup \mathcal{S}) \to 2^{Q \times (\Gamma \cup \mathcal{S}) \times \mathcal{H}}$ is a transition relation.*

Moreover, for any pair $(q, a) \in Q \times (\Gamma \cup \mathcal{S})$, each element $(q', a', d) \in \delta(q, a)$ satisfies:

- *$a \in \mathcal{S}$ implies $d = \nu(a)$ and $a' = a$, and*
- *$a \notin \mathcal{S}$ implies $a' \notin \mathcal{S}$.*

If for for each $q \in Q$ and $a \in \Gamma \cup \mathcal{S}$ we have $|\delta(q, a)| \leq 1$, we say that $\mathcal{A}$ is a deterministic bounded 2TM.

Let $P \in \Sigma^{*,*}$ be an input to a bounded 2TM *$\mathcal{A}$. In the initial configuration of $\mathcal{A}$ on P, its tape contains $\widehat{P}$, its control unit is in state q_0 and the head scans the top-left corner of P. When $P = \Lambda$, the head scans the bottom-right corner of $\widehat{P}$ containing $\#$. The machine accepts P iff there is a computation of $\mathcal{A}$ starting in the initial configuration on P and finishing in an accepting state from Q_F.*

Definition 7. *A* sgraffito automaton *(*2SA*) is a tuple*
$\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_0, Q_F, \mu)$ where

- *$(Q, \Sigma, \Gamma, \delta, q_0, Q_F)$ is a two-dimensional bounded Turing machine,*
- *$\mu : \Gamma \to \mathbb{N}$ is a weight function and*
- *the transition relation δ satisfies*

  $$(q', a', d) \in \delta(q, a) \wedge a \notin \mathcal{S} \implies \mu(a') < \mu(a) \text{ for all } q, q' \in Q, d \in \mathcal{H}, a, a' \in \Gamma \cup \mathcal{S}.$$

$\mathcal{A}$ is a deterministic 2SA*, if the underlying bounded* 2TM *$(Q, \Sigma, \Gamma, \delta, q_0, Q_F)$ is deterministic.*

We denote by $\mathcal{L}(\mathsf{2SA})$ the class of all languages accepted by 2SA and by $\mathcal{L}(\mathsf{2DSA})$ the class of all languages accepted by 2DSA.

From the same source ([17]) come the following closure properties of $\mathcal{L}(\mathsf{2SA})$ and $\mathcal{L}(\mathsf{2DSA})$, their relation to REC and DREC and also an important lemma that we often use when asserting that a certain sgraffito automaton can be constructed.

Theorem 1. *The class $\mathcal{L}(\mathsf{2SA})$ is closed under projection, rotation, mirroring, union, intersection and both row and column concatenation.*

Theorem 2. *The class $\mathcal{L}(\mathsf{2SA})$ is not closed under complement.*

Theorem 3. *The class $\mathcal{L}(\mathsf{2DSA})$ is closed under complement, rotation, mirroring, union and intersection.*

Theorem 4. *The class $\mathcal{L}(\mathsf{2DSA})$ is not closed under projection or either row or column concatenation.*

Theorem 5. *REC is included in $\mathcal{L}(\mathsf{2SA})$,* DREC *is included in $\mathcal{L}(\mathsf{2DSA})$.*

Theorem 6. *The classes $\mathcal{L}(\mathsf{2DSA})$ and* REC *are incomparable.*

Lemma 1. *Let $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, Q_F)$ be a two-dimensional bounded Turing machine. Let $k \in \mathbb{N}$ be an integer such that during each computation of $\mathcal{M}$ over any picture in $\Sigma^{*,*}$ each tape field is visited at most k times. Then, there is a* 2SA *$\mathcal{A}$ such that $L(\mathcal{A}) = L(\mathcal{M})$. Moreover, if $\mathcal{M}$ is deterministic, $\mathcal{A}$ is deterministic too.*

Sgraffito automata are rather well established in the hierarchy of other models. This as well as the closure properties of the model is significant for us because our new model is of (practically) the same power and is therefore going to inherit all of these features.

1.4 Two-dimensional restarting tiling automaton

The restarting tiling automata were the first attempt to generalize one-dimensional restarting automata to two-dimensional inputs.

There are several variants of models of one-dimensional restarting automata, first of which was introduced in [12]. Here we give a brief informal description of an *RRWW*-automaton ([13]) — one of the most general models of restarting automata.

An *RRWW*-automaton operates on a tape, which at the beginning contains an input word (from an input alphabet Σ) that is surrounded by a special symbols ¢ and \$ from the left and the right side, respectively. The automaton's head has a lookahead window of fixed size k. The automaton starts in a so-called *restarting configuration*, being in an initial state q_0 with its head positioned over the first symbol ¢ (i.e. the window contains ¢ and the first $k-1$ symbols of the input). Then, the automaton performs a sequence of operations described by a transition function δ.

Based on the current state q and the contents of the lookahead window u, the automaton can take one of the following actions in each step:

- Change its state to some q' and move the head by one field to the right.
- Change its state to q' and rewrite u to v, where v is a word containing symbols from a working alphabet $\Gamma \supseteq \Sigma$. It is required that v is shorter than u and also that the boundary symbols $¢$, $\$$ are kept in their respective positions. After the rewriting, the automaton's head is moved to the first symbol after u (or to the right sentinel $\$$, if it was present in u).
- Perform a restart, i.e. put itself in the restarting configuration.

The computation terminates when the automaton reaches one of its halting states. These are divided into accepting states and rejecting states.

The restarts divide the computation into *cycles*. In each cycle (except for the last one), exactly one rewriting operation must be performed. This means that after a rewriting is carried out, the rest of the word is only scanned in order to decide whether to restart or halt. A variant of *RRWW*-automaton that restarts immediately after each rewriting is called an *RWW*-automaton.

The two-dimensional restarting tiling automata (2RTA) were introduced in [2] and [17]. As with *RRWW*-automata, a restarting tiling automaton works in cycles. In every cycle, the automaton scans a boundary picture using a window of size $(2, 2)$ (a tile). The order in which the tiles of the picture are scanned is given by a so-called *scanning strategy*.

A scanning strategy is defined by a starting position (one of the four corners of a picture) and a partial function $f : \mathbb{N}_0^4 \to \mathbb{N}_0^2$ which, for a tile position (i, j) in a picture of size (m, n), yields a position $(i', j') = f(i, j, m, n)$ of the next tile to be scanned. The scanning strategy must be constructed so that every tile position of a picture is scanned exactly once.

One cycle of a 2RTA consists of scanning tiles of a boundary picture in the order given by its scanning strategy until a tile is found which can be rewritten according to some of the automaton's rewriting rules. Every rewriting changes exactly one symbol in the scanned tile and must not affect the boundaries.

After each rewriting, the automaton restarts immediately. If the whole picture is scanned without finding a rewritable tile, the automaton halts and if the resulting picture belongs to a supplied local language, it is accepted.

The finiteness of every computation is ensured similarly as with sgraffito automata — the symbols have assigned non-negative integer weights, and a symbol can only be rewritten to a symbol of lower weight.

Unlike an *RRWW*-automaton, 2RTA is a stateless machine. After the introduction of 2RTA, there were introduced another models of two-dimensional restarting automata — the two-dimensional ordered restarting automaton (see [15]) and the extended two-way ordered restarting automaton (see [14]), which use states and are rather more complex than 2RTA as a result. As these models were introduced only recently, we will not use them in this thesis.

Following this informal introduction we include a formal definition of 2RTA taken from [17].

Definition 8. *A* two-dimensional restarting tiling automaton*, referred to as* 2RTA*, is a 6-tuple $\mathcal{M} = (\Sigma, \Gamma, \Theta_f, \delta, \nu, \mu)$, where Σ is a finite input alphabet, Γ is a finite working alphabet ($\Gamma \supseteq \Sigma$), $\Theta_f \subseteq (\Gamma \cup \mathcal{S})^{2,2}$ is a set of accepting tiles, ν is a scanning strategy, $\mu : \Gamma \to \mathbb{N}$ is a weight function and $\delta \subseteq \{(U \to V) | U, V \in$*

$(\Gamma \cup \mathcal{S})^{2,2}\}$ *is a set of rewriting rules satisfying the condition that in every rule* $u \to v$ *only a single position of* u*, containing a symbol* s *from* Γ*, is changed into some* $t \in \Gamma$ *such that* $\mu(t) < \mu(s)$*.*

In the following definition, $\nu(r, m, n)$ denotes the sequence of the first r positions of a picture of size (m, n) visited according to the scanning strategy ν.

Definition 9. *Let* $\mathcal{M} = (\Sigma, \Gamma, \Theta_f, \delta, \nu, \mu)$ *be a two-dimensional restarting tiling automaton,* P_1*,* P_2 *be two pictures over the alphabet* Γ *of the same size* (m, n) *and* $\nu((m+1)(n+1), m, n) = (i_0, j_0), \ldots, (i_{(m+1)(n+1)-1}, j_{(m+1)(n+1)-1})$*. We say that the picture* P_1 *can be* directly reduced to the picture P_2*, denoted by* $P_1 \vdash_{\mathcal{M}} P_2$*, if there exists an integer* $s, 0 \leq s < (m+1)(n+1)$*, such that* $\widehat{P_1}(k, l) = \widehat{P_2}(k, l)$ *for all pairs of the indices* (k, l)*, where* $1 \leq k \leq rows(\widehat{P_1}), 1 \leq l \leq cols(\widehat{P_1})$ *except the pairs* $(i_s, j_s), (i_s, j_s+1), (i_s+1, j_s), (i_s+1, j_s+1)$ *and there exists a rule*

$$\begin{array}{|c|c|}\hline P_1(i_s, j_s) & P_1(i_s, j_s+1) \\ \hline P_1(i_s+1, j_s) & P_1(i_s+1, j_s+1) \\ \hline \end{array} \to \begin{array}{|c|c|}\hline P_2(i_s, j_s) & P_2(i_s, j_s+1) \\ \hline P_2(i_s+1, j_s) & P_2(i_s+1, j_s+1) \\ \hline \end{array} \text{ in } \delta.$$

Moreover, there is no rule in δ *that could be applied to any tile*

$$\begin{array}{|c|c|}\hline P_1(i_r, j_r) & P_1(i_r, j_r+1) \\ \hline P_1(i_r+1, j_r) & P_1(i_r+1, j_r+1) \\ \hline \end{array},$$

where $0 \leq r < s$*. We say that* P_1 *can be* reduced to P_2 *(denoted by* $P_1 \vdash^*_{\mathcal{M}} P_2$*) if there exists a sequence of reductions* $Q_1 \vdash_{\mathcal{M}} Q_2, Q_2 \vdash_{\mathcal{M}} Q_3, \ldots, Q_{n-1} \vdash_{\mathcal{M}} Q_n$*, where* $n \geq 1, Q_1 = P_1$ *and* $Q_n = P_2$*. Obviously, the relation* $\vdash^*_{\mathcal{M}}$ *is the reflexive and transitive closure of the relation* $\vdash_{\mathcal{M}}$*.*

Let $\mathcal{M} = (\Sigma, \Gamma, \Theta_f, \delta, \nu, \mu)$ *be a* 2RTA*. The* language accepted by M *is the set*

$$L(\mathcal{M}) = \{P \in \Sigma^{*,*} | \exists Q \in \Gamma^{*,*} : P \vdash^*_{\mathcal{M}} Q \text{ and } Q \in L(\Theta_f)\}.$$

Definition 10. *A* deterministic two-dimensional restarting automaton*, referred to as* 2DRTA*, is a* 2RTA $\mathcal{M} = (\Sigma, \Gamma, \Theta_f, \delta, \nu, \mu)$ *with the set of rewriting rules* δ *satisfying one additional condition that for every tile* $T \in (\Gamma \cup \mathcal{S})^{2,2}$ *there exists at most one rule with the left-hand side* T *in* δ*.*

We denote by $\mathcal{L}(\mathsf{2RTA})$ the class of all languages accepted by 2RTA and by $\mathcal{L}(\mathsf{2DRTA})$ the class of all languages accepted by 2DRTA.

The relation between restarting tiling automata and sgraffito automata was established in [17].

Proposition 1. $\mathcal{L}(\mathsf{2SA})$ *is included in* $\mathcal{L}(\mathsf{2RTA})$*,* $\mathcal{L}(\mathsf{2DSA})$ *is included in* $\mathcal{L}(\mathsf{2DRTA})$*.*

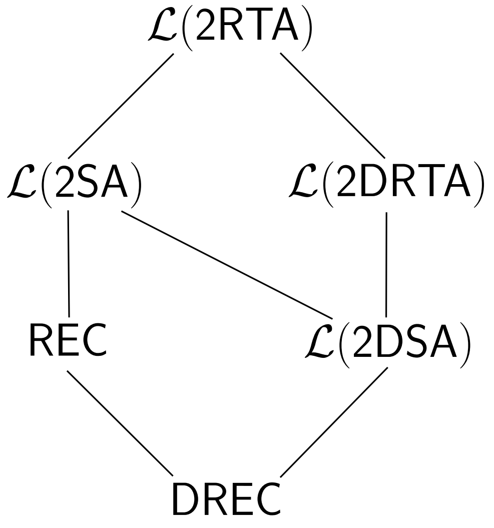

$$\begin{tikzcd}
 & \mathcal{L}(\mathsf{2RTA}) \arrow[dl, no head] \arrow[dr, no head] & \\
\mathcal{L}(\mathsf{2SA}) \arrow[d, no head] \arrow[drr, no head] & & \mathcal{L}(\mathsf{2DRTA}) \arrow[d, no head] \\
\mathsf{REC} \arrow[dr, no head] & & \mathcal{L}(\mathsf{2DSA}) \arrow[dl, no head] \\
 & \mathsf{DREC} &
\end{tikzcd}$$

Figure 1.1: Hierarchy of the presented classes of picture languages. Every line symbolizes an inclusion of the lower class in the upper class.

Chapter 2

Two-dimensional limited context restarting automaton

For the purpose of procedural generation of restarting automata, we now try to create a model whose definition is as simple as possible. The restarting tiling automaton appears to be a good starting point of our efforts.

To construct a 2RTA $\mathcal{M} = (\Sigma, \Gamma, \Theta_f, \delta, \nu, \mu)$, one needs to define several structures: a scanning strategy, a working alphabet with weights, a set of rewriting rules and a local language of accepted results. For our model we will try to reduce this list.

First candidate for reduction is the scanning strategy. Its variability may present some interesting possibilities, but here we will focus on shaping the automaton by the choice of rewriting rules, which renders the scanning strategy more of an impediment than help. Therefore we choose to omit the concept of scanning strategies altogether. From practical viewpoint, it means that one step of an automaton's computation is made simply by nondeterministically choosing any possible rewriting in the whole picture rather than looking for the first possibility in some given order. Alternatively, this can be interpreted as allowing the automaton to ignore a rewriting possibility and continue its scanning path (as it is with the *RRWW*-automaton, for example).

The obvious downside of this approach is of course the loss of an effective intuitive algorithm for deterministic simulation. To deal with this issue we propose an alternative recognition algorithm later in 5.1.

Without a scanning strategy, we lose the main reason why the rewriting rules come in the form of tiles. A rewriting is now simply an operation which is based on the contents of the rewritten field itself and its immediate neighborhood. To reflect that, we will consider the rules in the following form:

$$r = \begin{array}{|c|c|c|}\hline n_1 & n_2 & n_3 \\ \hline n_4 & s & n_5 \\ \hline n_6 & n_7 & n_8 \\ \hline \end{array} \rightarrow \begin{array}{|c|c|c|}\hline n_1 & n_2 & n_3 \\ \hline n_4 & s' & n_5 \\ \hline n_6 & n_7 & n_8 \\ \hline \end{array}$$

Or, for short

$$r = (s, s', (n_1, n_2, \ldots, n_8)) \in \Gamma \times \Gamma \times (\Gamma \cup \mathcal{S})^8.$$

This change generally does not alter the power of the automaton, because, as we will show later in 4.2, a conversion can be made both ways between the two formats.

Another simplification of the automaton can be done by fixing the local language Θ_f, or the condition of accepting a rewritten picture. One of the most natural conditions for accepting is that combined weight of the whole picture be zero. That means accepting pictures of the local language

$$\Theta_f^0 = (\mathcal{S} \cup \{a \mid \mu(a) = 0\})^{2,2}$$

A theoretical problem here might be that this gives the automaton no means to reject the empty picture Λ (when $\begin{array}{|c|c|}\hline \# & \# \\ \hline \# & \# \\ \hline \end{array} \notin \Theta_f$). For our purposes, this problem is inconsequential.

When not using a scanning strategy, this simplification does not at all alter the power of the automaton . Given an automaton accepting any local language Θ_f, another one can be constructed accepting the same language while accepting Θ_f^0.

This automaton rewrites a picture similarly to the original and then “freezes” the resulting picture by rewriting every symbol $a \in \Gamma$ to its frozen equivalent $a' \in \Gamma'$ using rules

$$r = \begin{array}{|c|c|c|}\hline n_1 & n_2 & n_3 \\ \hline n_4 & a & n_5 \\ \hline n_6 & n_7 & n_8 \\ \hline \end{array} \rightarrow \begin{array}{|c|c|c|}\hline n_1 & n_2 & n_3 \\ \hline n_4 & a' & n_5 \\ \hline n_6 & n_7 & n_8 \\ \hline \end{array}$$

for all $a \in \Gamma, n_1, \ldots, n_8 \in \Gamma \cup \Gamma' \cup \mathcal{S}$.

Symbols from Γ' cannot be further operated upon by the original rewriting rules. Without a scanning strategy, these rewritings are never forced to be done prematurely. Therefore any picture created using the original rules can be obtained in the frozen form.

The last stage of the computation is to verify that the tiles in the frozen parts of the picture belong to Θ_f. This can be done by rewriting the frozen symbols to their new zero-weight equivalents from Γ_0 by rules

$$r = \begin{array}{|c|c|c|}\hline n_1^0 & n_2^0 & n_3^0 \\ \hline n_4^0 & a' & n_5' \\ \hline n_6' & n_7' & n_8' \\ \hline \end{array} \rightarrow \begin{array}{|c|c|c|}\hline n_1^0 & n_2^0 & n_3^0 \\ \hline n_4^0 & a^0 & n_5' \\ \hline n_6' & n_7' & n_8' \\ \hline \end{array}$$

where for $n_1, \ldots, n_8, a \in \Gamma$, the original counterparts of $n_1^0, \ldots, n_4^0 \in \Gamma_0, n_5', \ldots, n_8', a' \in \Gamma'$ (assuming $s = s' = s^0$ for $s \in \mathcal{S}$), we have

$$\left\{ \begin{array}{|c|c|}\hline n_1 & n_2 \\ \hline n_4 & a \\ \hline \end{array}, \begin{array}{|c|c|}\hline n_2 & n_3 \\ \hline a & n_5 \\ \hline \end{array}, \begin{array}{|c|c|}\hline n_4 & a \\ \hline n_6 & n_7 \\ \hline \end{array}, \begin{array}{|c|c|}\hline a & n_5 \\ \hline n_7 & n_8 \\ \hline \end{array} \right\} \subseteq \Theta_f.$$

Note that the automaton has somewhat loose requirements for ordering of the performed operations. In a successful computation, some parts of a picture can already be frozen and even verified for locality before the original computation is concluded everywhere else. However, the freezing of the symbols ensures that the two phases of the computation never interfere.

To sum up, a formal definition (which is a modification of a 2RTA definition from [17]) follows.

2.1 Formal definition

Definition 11. *A* two-dimensional limited context restarting automaton*, referred to as* 2LCRA*, is a quadruple $\mathcal{M} = (\Sigma, \Gamma, \delta, \mu)$, where Σ is a finite input alphabet, Γ is a finite working alphabet ($\Gamma \supseteq \Sigma$), $\mu : \Gamma \to \mathbb{N}_0$ is a weight function and $\delta \subseteq \{(s, s', N) | s, s' \in \Gamma, N \in (\Gamma \cup \mathcal{S})^8\}$ is a set of rewriting rules satisfying the condition $(\forall (s, s', N) \in \delta)\mu(s') < \mu(s)$.*

Definition 12. *Let $\mathcal{M} = (\Sigma, \Gamma, \delta, \mu)$ be a two-dimensional limited context restarting automaton, P_1, P_2 be two pictures over the alphabet Γ of the same size (m, n). We say that the picture P_1 can be* directly reduced to the picture P_2*, denoted by $P_1 \vdash_{\mathcal{M}} P_2$, if there exists a pair $(i, j) \in \{1, \ldots, m\} \times \{1, \ldots, n\}$, such that $P_1(k, l) = P_2(k, l)$ for all pairs $(k, l) \in \{1, \ldots, m\} \times \{1, \ldots, n\} \setminus \{(i, j)\}$ and there exists a rule*

$$\begin{array}{|c|c|c|}\hline \widehat{P}_1(i-1,j-1) & \widehat{P}_1(i-1,j) & \widehat{P}_1(i-1,j+1) \\ \hline \widehat{P}_1(i,j-1) & \widehat{P}_1(i,j) & \widehat{P}_1(i,j+1) \\ \hline \widehat{P}_1(i+1,j-1) & \widehat{P}_1(i+1,j) & \widehat{P}_1(i+1,j+1) \\ \hline \end{array}$$

$$\downarrow$$

$$\begin{array}{|c|c|c|}\hline \widehat{P}_1(i-1,j-1) & \widehat{P}_1(i-1,j) & \widehat{P}_1(i-1,j+1) \\ \hline \widehat{P}_1(i,j-1) & \widehat{P}_2(i,j) & \widehat{P}_1(i,j+1) \\ \hline \widehat{P}_1(i+1,j-1) & \widehat{P}_1(i+1,j) & \widehat{P}_1(i+1,j+1) \\ \hline \end{array}$$

in δ.

We say that P_1 can be reduced to P_2 *(denoted by $P_1 \vdash^*_{\mathcal{M}} P_2$) if there exists a sequence of reductions $Q_1 \vdash_{\mathcal{M}} Q_2, Q_2 \vdash_{\mathcal{M}} Q_3, \ldots, Q_{n-1} \vdash_{\mathcal{M}} Q_n$, where $n \geq 1$, $Q_1 = P_1$ and $Q_n = P_2$. Obviously, the relation $\vdash^*_{\mathcal{M}}$ is the reflexive and transitive closure of the relation $\vdash_{\mathcal{M}}$.*

Let $\mathcal{M} = (\Sigma, \Gamma, \delta, \mu)$ be a 2LCRA*. We refer to the symbols in $\Gamma \setminus \Sigma$ as auxiliary symbols. Furthermore, we denote by Γ_0 the set $\{\gamma \mid \gamma \in \Gamma, \mu(\gamma) = 0\}$ and we say that a picture $P \in \Gamma^{*,*}$ can be rewritten to zero by $\mathcal{M}$ if $(\exists Q \in \Gamma_0^{*,*}), P \vdash^*_{\mathcal{M}} Q$. The* language accepted by $\mathcal{M}$ *is the set*

$$L(\mathcal{M}) = \{P \in \Sigma^{*,*} \mid (\exists Q \in \Gamma_0^{*,*})\, P \vdash^*_{\mathcal{M}} Q\}.$$

*We denote by $\mathcal{L}$(*2LCRA*) the class of all languages accepted by* 2LCRA*.*

2.2 Correctness Preservation

The existing models usually have a deterministic variant which yields a subclass of automata that can be simulated in polynomial time. Our model does not allow for any reasonable determinism, but we can substitute it by the notion of correctness preservation.

For this concept we take inspiration from the correctness preserving property that was introduced for one-dimensional restarting automata (see e.g. [16]).

A restarting automaton is correctness preserving if every rewriting of an accepted word yields another accepted word. It means that the automaton is not capable of making a mistake that would result in false rejection.

We can define this property of inability to make mistakes for our model as well.

Definition 13. *Let $\mathcal{M} = (\Sigma, \Gamma, \delta, \mu)$ be a two-dimensional limited context restarting automaton. $\mathcal{M}$ is* correctness preserving *(referred to as* CP2LCRA*) if*

$$(\forall P \in \Gamma^{*,*})\,((\exists P_0 \in \Gamma_0^{*,*})\, P \vdash_{\mathcal{M}}^{*} P_0) \implies$$
$$(\forall P' \in \Gamma^{*,*}, P \vdash_{\mathcal{M}} P')\,(\exists P_0' \in \Gamma_0^{*,*})\, P \vdash_{\mathcal{M}}^{*} P_0'$$

where $\Gamma_0 = \{a | a \in \Gamma, \mu(a) = 0\}$.

*We denote by $\mathcal{L}$(*CP2LCRA*) the class of all languages accepted by* CP2LCRA.

In other words, with a CP2LCRA, if a picture can be rewritten to zero, then it can be rewritten to zero even after any valid application of a rewriting rule.

For an automaton $\mathcal{M}$ without any auxiliary symbols ($\Gamma = \Sigma$), the correctness preservation property can be formulated simply as

$$(\forall P \in L(\mathcal{M}), Q \in \Sigma^{*,*}) P \vdash_{\mathcal{M}}^{*} Q \implies Q \in L(\mathcal{M}).$$

A correctness preserving automaton can be simulated in polynomial time because we can in every step choose to apply any possible rewriting. Our choice cannot affect the final outcome, so there is no need to backtrack.

The downside of this definition is that for an arbitrary automaton it is likely undecidable whether it is correctness preserving.

As a simple example of a CP2LCRA, consider an automaton $\mathcal{M} = (\Sigma, \Sigma, \delta, \mu)$ that recognizes pictures over $\Sigma = \{\mathbf{a}, \mathbf{b}\}$ that only contain $\mathbf{b}$'s in horizontal lines that do not touch borders of the picture or each other (see example in Fig. 2.1). The weight function is $\mu(\mathbf{b}) = 1$, $\mu(\mathbf{a}) = 0$ and δ contains rules

$$\begin{array}{|c|c|c|}\hline \mathbf{a}&\mathbf{a}&\mathbf{a}\\\hline \mathbf{a}&\mathbf{b}&\mathbf{b}\\\hline \mathbf{a}&\mathbf{a}&\mathbf{a}\\\hline\end{array} \rightarrow \begin{array}{|c|c|c|}\hline \mathbf{a}&\mathbf{a}&\mathbf{a}\\\hline \mathbf{a}&\mathbf{a}&\mathbf{b}\\\hline \mathbf{a}&\mathbf{a}&\mathbf{a}\\\hline\end{array}$$
$$\begin{array}{|c|c|c|}\hline \mathbf{a}&\mathbf{a}&\mathbf{a}\\\hline \mathbf{a}&\mathbf{b}&\mathbf{a}\\\hline \mathbf{a}&\mathbf{a}&\mathbf{a}\\\hline\end{array} \rightarrow \begin{array}{|c|c|c|}\hline \mathbf{a}&\mathbf{a}&\mathbf{a}\\\hline \mathbf{a}&\mathbf{a}&\mathbf{a}\\\hline \mathbf{a}&\mathbf{a}&\mathbf{a}\\\hline\end{array}.$$

Each rewriting of a picture P to P' performed by $\mathcal{M}$ either shortens a horizontal line of $\mathbf{b}$'s or erases a line of unit length. Obviously, if P was from $L(\mathcal{M})$, then P' is also from $L(\mathcal{M})$. A successful computation ends with a picture from $\{\mathbf{a}\}^{*,*}$.

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

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

Figure 2.1: An example of a picture that belongs to the language of horizontal lines (left) and of a picture that does not (right).

Chapter 3

Language Examples

In this chapter, we will list some picture languages mostly taken from various literature, together with descriptions of 2LCRA accepting them. The presented automata are fairly simple so the usage of our model is well justified. This simplicity also gives us hope that languages like these can be learned by a generic learning algorithm.

To describe rewriting rules we will use rule patterns

$$\begin{array}{|c|c|c|}\hline N_1 & N_2 & N_3 \\ \hline N_4 & s & N_5 \\ \hline N_6 & N_7 & N_8 \\ \hline \end{array} \to \begin{array}{|c|c|c|}\hline N_1 & N_2 & N_3 \\ \hline N_4 & s' & N_5 \\ \hline N_6 & N_7 & N_8 \\ \hline \end{array},$$

(where $s, s' \in \Gamma$ are symbols and $N_1, N_2, \ldots, N_8 \subseteq \Gamma \cup \mathcal{S}$ are sets or lists of symbols) which denote sets of rules

$$\begin{array}{|c|c|c|}\hline n_1 & n_2 & n_3 \\ \hline n_4 & s & n_5 \\ \hline n_6 & n_7 & n_8 \\ \hline \end{array} \to \begin{array}{|c|c|c|}\hline n_1 & n_2 & n_3 \\ \hline n_4 & s' & n_5 \\ \hline n_6 & n_7 & n_8 \\ \hline \end{array},$$

where $n_1 \in N_1, n_2 \in N_2, \ldots, n_8 \in N_8$.

If any N_i contains all working symbols as well as the sentinels, we may omit the respective cell. For better clarity, we highlight the rewritten field by gray background.

3.1 Square picture

The first example is a very simple language that uses a single letter alphabet. It contains exactly all pictures P such that $\text{cols}(P) = \text{rows}(P)$, that is,

$$L = \bigcup_{n \in \mathbb{N}_0} \{\mathbf{a}\}^{n,n}.$$

This language can be found in many articles, for example in the *collection of examples* in [8].

As usual, our automaton can recognize such pictures by drawing a main diagonal that has to end in the bottom right corner:

$$\begin{array}{|c|c|}\hline \# & \top \\ \hline \vdash & \mathbf{a} \\ \hline \end{array} \to \begin{array}{|c|c|}\hline \# & \top \\ \hline \vdash & \mathbf{1} \\ \hline \end{array}, \quad \begin{array}{|c|c|}\hline \mathbf{1} & \mathbf{a} \\ \hline \mathbf{a} & \mathbf{a} \\ \hline \end{array} \to \begin{array}{|c|c|}\hline \mathbf{1} & \mathbf{a} \\ \hline \mathbf{a} & \mathbf{1} \\ \hline \end{array}$$

After the diagonal is drawn, all positions to the left and top of the diagonal are rewritten to zero:

$$\begin{array}{|c|c|}\hline \mathbf{a} & \mathbf{0,1} \\ \hline \end{array} \rightarrow \begin{array}{|c|c|}\hline \mathbf{0} & \mathbf{0,1} \\ \hline \end{array}, \quad \begin{array}{|c|}\hline \mathbf{a} \\ \hline \mathbf{0,1} \\ \hline \end{array} \rightarrow \begin{array}{|c|}\hline \mathbf{0} \\ \hline \mathbf{0,1} \\ \hline \end{array},$$

The weight function is $\mu(\mathbf{a}) = 1, \mu(\mathbf{1}) = 0, \mu(\mathbf{0}) = 0$.

1	0	0	a	a	a
0	1	0	0	a	a
a	a	1	0	a	a
a	0	a	1	a	a
a	0	0	0	1	a
a	a	a	a	a	a

Figure 3.1: An example of a possible rewriting of a square picture. Any of the marked cells can be rewritten in the next step.

Note that the actual rewriting of an input does not need to happen in the outlined order. The writing of zeros may commence before the whole diagonal is drawn. The rules might also suggest that the zeros are only propagated horizontally below the diagonal and vertically above it, but that is not the case. Zeros can always propagate both to the left and to the top (see Fig. 3.1).

However, when a position (i, j) is zeroed, there is always a diagonal cell $(k, k), k \geq i, j$ that has already been rewritten to $\mathbf{1}$. Therefore, in a non-square picture, positions $(\{i, j\})$ where $\max(i, j) > \min(\{\text{cols}(P), \text{rows}(P)\})$ can never be zeroed.

3.2 Permutation

The language of permutations is an often used example of a recognizable picture language (for example in [18]). A picture over a two letter alphabet $\{\mathbf{a}, \mathbf{b}\}$ represents a permutation if each row and each column contains exactly one letter $\mathbf{b}$. A 2LCRA recognizing such pictures performs the following operations:

- If $\mathbf{b}$ is not adjacent (vertically or horizontally) to another $\mathbf{b}$, it is rewritten to a zero-weight symbol $\mathbf{b_0}$.

$$\begin{array}{ccc}\cline{2-2} & \multicolumn{1}{|c|}{\mathbf{a}, \top} & \\ \hline \multicolumn{1}{|c|}{\mathbf{a}, \vdash} & \mathbf{b} & \multicolumn{1}{|c|}{\mathbf{a}, \dashv} \\ \hline & \multicolumn{1}{|c|}{\mathbf{a}, \bot} & \\ \cline{2-2} \end{array} \rightarrow \begin{array}{ccc}\cline{2-2} & \multicolumn{1}{|c|}{\mathbf{a}, \top} & \\ \hline \multicolumn{1}{|c|}{\mathbf{a}, \vdash} & \mathbf{b_0} & \multicolumn{1}{|c|}{\mathbf{a}, \dashv} \\ \hline & \multicolumn{1}{|c|}{\mathbf{a}, \bot} & \\ \cline{2-2} \end{array}$$

- A horizontal wave of rewriting of symbols $\mathbf{a}$ to $\mathbf{a_1}$ is sent from the $\mathbf{b_0}$'s. This wave doesn't continue if a second $\mathbf{b}$ is encountered.

  The wave is sent to the left using the rules

$$\begin{array}{|c|c|c|}\hline \mathbf{a}, \vdash & \mathbf{a} & \mathbf{a_1}, \mathbf{b_0} \\ \hline \end{array} \rightarrow \begin{array}{|c|c|c|}\hline \mathbf{a}, \vdash & \mathbf{a_1} & \mathbf{a_1}, \mathbf{b_0} \\ \hline \end{array}$$

and to the right using the rules

$$\begin{array}{|c|c|c|}\hline \mathbf{a_1},\mathbf{b_0} & \mathbf{a} & \mathbf{a},\vdash \\ \hline\end{array} \rightarrow \begin{array}{|c|c|c|}\hline \mathbf{a_1},\mathbf{b_0} & \mathbf{a_1} & \mathbf{a},\vdash \\ \hline\end{array}$$

- Then the same procedure is used to verify the number of $\mathbf{b}$'s in each column. Now the symbol $\mathbf{a_1}$ is rewritten to a zero-weight symbol $\mathbf{a_0}$:

$$\begin{array}{|c|}\hline \mathbf{a_1},\top \\ \hline \mathbf{a_1} \\ \hline \mathbf{a_0},\mathbf{b_0} \\ \hline\end{array} \rightarrow \begin{array}{|c|}\hline \mathbf{a_1},\top \\ \hline \mathbf{a_0} \\ \hline \mathbf{a_0},\mathbf{b_0} \\ \hline\end{array}, \quad \begin{array}{|c|}\hline \mathbf{a_0},\mathbf{b_0} \\ \hline \mathbf{a_1} \\ \hline \mathbf{a_1},\bot \\ \hline\end{array} \rightarrow \begin{array}{|c|}\hline \mathbf{a_0},\mathbf{b_0} \\ \hline \mathbf{a_0} \\ \hline \mathbf{a_1},\bot \\ \hline\end{array}$$

The weight function μ is defined as $\mu(\mathbf{a}) = \mu(\mathbf{b}) = 2, \mu(\mathbf{a_1}) = 1, \mu(\mathbf{a_0}) = \mu(\mathbf{b_0}) = 0$.

Like in the previous language, the order of rewriting is not fixed, but the operations always retain their meaning.

3.3 The First Column Equals Some Column

A set of pictures where the first column is equal to some other column is another popular recognizable picture language (found for example in [8]). We will assume a two-letter input alphabet $\{\mathbf{a}, \mathbf{b}\}$ for the sake of simplicity.

A 2LCRA recognizing this language can be implemented to announce the contents of the first column to the whole picture

$$\begin{array}{lll}
\begin{array}{|c|c|}\hline \vdash & \mathbf{a} \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline \vdash & \mathbf{a_a} \\ \hline\end{array} & & \\
\begin{array}{|c|c|}\hline \mathbf{a_a},\mathbf{b_a} & \mathbf{a} \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline \mathbf{a_a},\mathbf{b_a} & \mathbf{a_a} \\ \hline\end{array}, & & \begin{array}{|c|c|}\hline \mathbf{a_a},\mathbf{b_a} & \mathbf{b} \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline \mathbf{a_a},\mathbf{b_a} & \mathbf{b_a} \\ \hline\end{array} \\
\begin{array}{|c|c|}\hline \vdash & \mathbf{b} \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline \vdash & \mathbf{b_b} \\ \hline\end{array} & & \\
\begin{array}{|c|c|}\hline \mathbf{a_b},\mathbf{b_b} & \mathbf{a} \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline \mathbf{a_b},\mathbf{b_b} & \mathbf{a_b} \\ \hline\end{array}, & & \begin{array}{|c|c|}\hline \mathbf{a_b},\mathbf{b_b} & \mathbf{b} \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline \mathbf{a_b},\mathbf{b_b} & \mathbf{b_b} \\ \hline\end{array}
\end{array}$$

and then check any number of columns for equality (from top to bottom), rewriting the equal parts to an auxiliary symbol $\mathbf{1}$.

$$\begin{array}{|c|}\hline \top \\ \hline \mathbf{a_a} \\ \hline\end{array} \rightarrow \begin{array}{|c|}\hline \top \\ \hline \mathbf{1} \\ \hline\end{array}, \quad \begin{array}{|c|}\hline \top \\ \hline \mathbf{b_b} \\ \hline\end{array} \rightarrow \begin{array}{|c|}\hline \top \\ \hline \mathbf{1} \\ \hline\end{array}$$

$$\begin{array}{|c|}\hline \mathbf{1} \\ \hline \mathbf{a_a} \\ \hline\end{array} \rightarrow \begin{array}{|c|}\hline \mathbf{1} \\ \hline \mathbf{1} \\ \hline\end{array}, \quad \begin{array}{|c|}\hline \mathbf{1} \\ \hline \mathbf{b_b} \\ \hline\end{array} \rightarrow \begin{array}{|c|}\hline \mathbf{1} \\ \hline \mathbf{1} \\ \hline\end{array}$$

Finally if a whole column is equal to the first one (and it is not the first column itself), a symbol $\mathbf{0}$ is spawned.

$$\begin{array}{|c|c|}\hline s & \mathbf{1} \\ \hline \bot & \bot \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline s & \mathbf{0} \\ \hline \bot & \bot \\ \hline\end{array}$$

where $s \in \{\mathbf{a_a}, \mathbf{b_b}, \mathbf{a_b}, \mathbf{b_a}, \mathbf{1}\}$ The zero-weight symbol $\mathbf{0}$ is then indiscriminately spread throughout the whole picture.

The weights of the working symbols are $\mu(\mathbf{a}) = \mu(\mathbf{b}) = 3, \mu(\mathbf{a_a}) = \mu(\mathbf{a_b}) = \mu(\mathbf{b_a}) = \mu(\mathbf{b_b}) = 2, \mu(\mathbf{1}) = 1$ and $\mu(\mathbf{0}) = 0$.

3.4 Forest

A picture $P \in \{\mathbf{a}, \mathbf{b}\}^{*,*}$ can represent an unoriented graph $G(P) = (V, E)$ where

$$\begin{aligned} V &= \{(i,j) \mid P(i,j) = \mathbf{b}\} \\ E &= (V \times V) \cap \{((i,j),(i',j')) \mid |i-i'| + |j-j'| = 1\} \end{aligned}$$

The language of forests is then the set of all pictures P such that $G(P)$ is an acyclic graph (for example in Fig. 3.2). Similar language of trees is mentioned for example in [2].

A limited context automaton can process such pictures by erasing leaves of the depicted trees, until there are none left. If a non-empty graph does not have any leaves, then it necessarily contains a cycle.

A leaf can be matched and erased by one of the following rules:

$$\begin{array}{|c|c|c|} \hline & \top, \mathbf{a} & \\ \hline \vdash, \mathbf{a} & \mathbf{b} & \dashv, \mathbf{a}, \mathbf{b} \\ \hline & \bot, \mathbf{a} & \\ \hline \end{array} \rightarrow \begin{array}{|c|c|c|} \hline & \top, \mathbf{a} & \\ \hline \vdash, \mathbf{a} & \mathbf{a} & \dashv, \mathbf{a}, \mathbf{b} \\ \hline & \bot, \mathbf{a} & \\ \hline \end{array}$$

$$\begin{array}{|c|c|c|} \hline & \top, \mathbf{a}, \mathbf{b} & \\ \hline \vdash, \mathbf{a} & \mathbf{b} & \dashv, \mathbf{a} \\ \hline & \bot, \mathbf{a} & \\ \hline \end{array} \rightarrow \begin{array}{|c|c|c|} \hline & \top, \mathbf{a}, \mathbf{b} & \\ \hline \vdash, \mathbf{a} & \mathbf{a} & \dashv, \mathbf{a} \\ \hline & \bot, \mathbf{a} & \\ \hline \end{array}$$

$$\begin{array}{|c|c|c|} \hline & \top, \mathbf{a} & \\ \hline \vdash, \mathbf{a}, \mathbf{b} & \mathbf{b} & \dashv, \mathbf{a} \\ \hline & \bot, \mathbf{a} & \\ \hline \end{array} \rightarrow \begin{array}{|c|c|c|} \hline & \top, \mathbf{a} & \\ \hline \vdash, \mathbf{a}, \mathbf{b} & \mathbf{a} & \dashv, \mathbf{a} \\ \hline & \bot, \mathbf{a} & \\ \hline \end{array}$$

$$\begin{array}{|c|c|c|} \hline & \top, \mathbf{a} & \\ \hline \vdash, \mathbf{a} & \mathbf{b} & \dashv, \mathbf{a} \\ \hline & \bot, \mathbf{a}, \mathbf{b} & \\ \hline \end{array} \rightarrow \begin{array}{|c|c|c|} \hline & \top, \mathbf{a} & \\ \hline \vdash, \mathbf{a} & \mathbf{a} & \dashv, \mathbf{a} \\ \hline & \bot, \mathbf{a}, \mathbf{b} & \\ \hline \end{array}$$

The result of a successful computation is a picture without a single **b**, which induces the weights $\mu(\mathbf{b}) = 1$ and $\mu(\mathbf{a}) = 0$.

The way in which this automaton operates (rewriting a forest to obtain a smaller forest) naturally makes it correctness preserving.

a	**a**	**a**	**a**	**b**	**b**	**b**
a	**b**	**a**	**b**	**a**	**a**	**b**
b	**b**	**b**	**b**	**a**	**a**	**b**
a	**b**	**a**	**a**	**a**	**b**	**b**
a	**b**	**b**	**b**	**b**	**b**	**a**
a	**a**	**a**	**a**	**b**	**a**	**a**
a	**b**	**a**	**b**	**b**	**b**	**b**
a	**b**	**a**	**a**	**a**	**a**	**a**

a	**a**	**a**	**b**	**b**	**b**	**b**
a	**b**	**a**	**b**	**a**	**a**	**b**
b	**b**	**b**	**b**	**a**	**a**	**b**
a	**b**	**a**	**a**	**a**	**a**	**b**
a	**b**	**b**	**b**	**b**	**b**	**b**
a	**a**	**a**	**a**	**b**	**a**	**b**
a	**b**	**a**	**b**	**b**	**b**	**b**
a	**b**	**a**	**a**	**a**	**a**	**a**

Figure 3.2: A picture that represents a forest (left) and a picture that does not (right).

3.5 Single Object

Using the same definition of a graph represented by a given picture as above we can define the language of pictures depicting a single object as the set of those pictures P where $G(P)$ has at most one component.

In the spirit of correctness preservation, we can build an automaton that transforms a picture by gradually growing an object until it fills the whole picture. The rewriting rules facilitating this growth must ensure that two objects are not connected by the rewriting. For example, rules like

$$\begin{array}{|c|c|c|}\hline \mathbf{b}&\mathbf{a}&\mathbf{a}\\\hline \mathbf{b}&\mathbf{a}&\mathbf{b}\\\hline \mathbf{b}&\mathbf{b}&\mathbf{b}\\\hline\end{array} \rightarrow \begin{array}{|c|c|c|}\hline \mathbf{b}&\mathbf{a}&\mathbf{a}\\\hline \mathbf{b}&\mathbf{b}&\mathbf{b}\\\hline \mathbf{b}&\mathbf{b}&\mathbf{b}\\\hline\end{array} \quad \text{or} \quad \begin{array}{|c|c|c|}\hline \mathbf{b}&\mathbf{b}&\mathbf{a}\\\hline \mathbf{a}&\mathbf{a}&\mathbf{a}\\\hline \mathbf{a}&\mathbf{a}&\mathbf{a}\\\hline\end{array} \rightarrow \begin{array}{|c|c|c|}\hline \mathbf{b}&\mathbf{b}&\mathbf{a}\\\hline \mathbf{a}&\mathbf{b}&\mathbf{a}\\\hline \mathbf{a}&\mathbf{a}&\mathbf{a}\\\hline\end{array}$$

can never connect two disjoint objects whereas rules like

$$\begin{array}{|c|c|c|}\hline \mathbf{b}&\mathbf{a}&\mathbf{a}\\\hline \mathbf{b}&\mathbf{a}&\mathbf{b}\\\hline \mathbf{a}&\mathbf{b}&\mathbf{b}\\\hline\end{array} \rightarrow \begin{array}{|c|c|c|}\hline \mathbf{b}&\mathbf{a}&\mathbf{a}\\\hline \mathbf{b}&\mathbf{b}&\mathbf{b}\\\hline \mathbf{a}&\mathbf{b}&\mathbf{b}\\\hline\end{array} \quad \text{or} \quad \begin{array}{|c|c|c|}\hline \mathbf{b}&\mathbf{b}&\mathbf{a}\\\hline \mathbf{a}&\mathbf{a}&\mathbf{a}\\\hline \mathbf{a}&\mathbf{b}&\mathbf{a}\\\hline\end{array} \rightarrow \begin{array}{|c|c|c|}\hline \mathbf{b}&\mathbf{b}&\mathbf{a}\\\hline \mathbf{a}&\mathbf{b}&\mathbf{a}\\\hline \mathbf{a}&\mathbf{b}&\mathbf{a}\\\hline\end{array}$$

might do that and therefore are not contained in δ.

Unlike the previous example, the zero-weight symbol here is **b**.

Using the correctness preservation property, this automaton can be simulated efficiently in linear time. On the other hand, simulating it as an arbitrary automaton is very inefficient. Given a picture with two objects, the simulator might have to explore many dead-end computations where the picture is almost filled and only a little gap is left between the two grown objects.

3.6 SAT

To illustrate NP-completeness of the recognition of languages in $\mathcal{L}(\mathsf{2LCRA})$ we will describe a language of pictures that represent satisfiable Boolean formulas in conjunctive normal form (CNF).

An input CNF formula

$$\bigwedge_{i=1}^{m} \bigvee_{j_i=1}^{n_i} p_{j_i}$$

with a total of n variables $x_1, ..., x_n$ can be represented by a picture $P \in \{\mathbf{T}, \mathbf{F}, \mathbf{N}\}^{m,n}$ where

$$P(i,j) = \begin{cases} \mathbf{T} & \text{if } i\text{-th clause contains } x_j, \\ \mathbf{F} & \text{if } i\text{-th clause contains } \neg x_j, \\ \mathbf{N} & \text{otherwise.} \end{cases}$$

These pictures are recognized by an automaton that uses following rules:

First, a value (true or false) is selected for each variable (column) and added to the respective symbol as a subscript.

$$\begin{array}{|c|}\hline \top \\ \hline \mathbf{T} \\ \hline\end{array} \rightarrow \begin{array}{|c|}\hline \top \\ \hline \mathbf{T_T} \\ \hline\end{array}, \quad \begin{array}{|c|}\hline \top \\ \hline \mathbf{T} \\ \hline\end{array} \rightarrow \begin{array}{|c|}\hline \top \\ \hline \mathbf{T_F} \\ \hline\end{array}$$

$$\begin{array}{|c|}\hline \top \\ \hline \mathbf{F} \\ \hline\end{array} \rightarrow \begin{array}{|c|}\hline \top \\ \hline \mathbf{F_T} \\ \hline\end{array}, \quad \begin{array}{|c|}\hline \top \\ \hline \mathbf{F} \\ \hline\end{array} \rightarrow \begin{array}{|c|}\hline \top \\ \hline \mathbf{F_F} \\ \hline\end{array}$$

$$\begin{array}{|c|}\hline \top \\ \hline \mathbf{N} \\ \hline\end{array} \rightarrow \begin{array}{|c|}\hline \top \\ \hline \mathbf{N_T} \\ \hline\end{array}, \quad \begin{array}{|c|}\hline \top \\ \hline \mathbf{N} \\ \hline\end{array} \rightarrow \begin{array}{|c|}\hline \top \\ \hline \mathbf{N_F} \\ \hline\end{array}$$

This value is then propagated down the whole column.

$$\begin{array}{|c|}\hline \mathbf{T_T}, \mathbf{F_T}, \mathbf{N_T} \\ \hline \mathbf{T} \\ \hline\end{array} \rightarrow \begin{array}{|c|}\hline \mathbf{T_T}, \mathbf{F_T}, \mathbf{N_T} \\ \hline \mathbf{T_T} \\ \hline\end{array}, \quad \begin{array}{|c|}\hline \mathbf{T_F}, \mathbf{F_F}, \mathbf{N_F} \\ \hline \mathbf{T} \\ \hline\end{array} \rightarrow \begin{array}{|c|}\hline \mathbf{T_F}, \mathbf{F_F}, \mathbf{N_F} \\ \hline \mathbf{T_F} \\ \hline\end{array}$$

$$\begin{array}{|c|}\hline \mathbf{T_T}, \mathbf{F_T}, \mathbf{N_T} \\ \hline \mathbf{F} \\ \hline\end{array} \rightarrow \begin{array}{|c|}\hline \mathbf{T_T}, \mathbf{F_T}, \mathbf{N_T} \\ \hline \mathbf{F_T} \\ \hline\end{array}, \quad \begin{array}{|c|}\hline \mathbf{T_F}, \mathbf{F_F}, \mathbf{N_F} \\ \hline \mathbf{F} \\ \hline\end{array} \rightarrow \begin{array}{|c|}\hline \mathbf{T_F}, \mathbf{F_F}, \mathbf{N_F} \\ \hline \mathbf{F_F} \\ \hline\end{array}$$

$$\begin{array}{|c|}\hline \mathbf{T_T}, \mathbf{F_T}, \mathbf{N_T} \\ \hline \mathbf{N} \\ \hline\end{array} \rightarrow \begin{array}{|c|}\hline \mathbf{T_T}, \mathbf{F_T}, \mathbf{N_T} \\ \hline \mathbf{N_T} \\ \hline\end{array}, \quad \begin{array}{|c|}\hline \mathbf{T_F}, \mathbf{F_F}, \mathbf{N_F} \\ \hline \mathbf{N} \\ \hline\end{array} \rightarrow \begin{array}{|c|}\hline \mathbf{T_F}, \mathbf{F_F}, \mathbf{N_F} \\ \hline \mathbf{N_F} \\ \hline\end{array}$$

If the guessed assignment satisfies the formula, then for each clause (row) at least one atom is found that makes it true:

$$\boxed{\mathbf{T_T}} \rightarrow \boxed{\mathbf{0}}, \quad \boxed{\mathbf{F_F}} \rightarrow \boxed{\mathbf{0}}$$

Finally the zeros are propagated horizontally, making the whole rows that represent satisfied clauses zero. If the chosen assignment does not satisfy some clauses (and thus the whole formula), then the corresponding rows are left non-zero (because zero cannot appear in them).

Chapter 4

2LCRA properties

In this chapter we will explore the class of languages accepted by 2LCRA. First we will prove the common closure properties and then we will insert our model into the hierarchy of existing two-dimensional models.

4.1 Closure properties

Here we will show basic closure properties of our model. Many of them can be proved similarly as with 2RTA (see [2]). These properties can also be deduced from the fact that 2LCRA have virtually the same power as 2SA (see 4.3.2).

Observation 1. *The class* $\mathcal{L}(\mathsf{2LCRA})$ *is closed under rotation and horizontal and vertical mirroring.*

Proof. Given an automaton $\mathcal{M}$ accepting the language $L(\mathcal{M})$ we can construct an automaton accepting rotation or mirroring of $L(\mathcal{M})$ by simply replacing all the rewriting rules by their rotated or mirrored versions.

In the case of clockwise rotation, a rule

$$\begin{array}{|c|c|c|}\hline n_1 & n_2 & n_3 \\ \hline n_4 & s & n_5 \\ \hline n_6 & n_7 & n_8 \\ \hline\end{array} \rightarrow \begin{array}{|c|c|c|}\hline n_1 & n_2 & n_3 \\ \hline n_4 & s' & n_5 \\ \hline n_6 & n_7 & n_8 \\ \hline\end{array}$$

is replaced by a rule

$$\begin{array}{|c|c|c|}\hline f(n_6) & f(n_4) & f(n_1) \\ \hline f(n_7) & s & f(n_2) \\ \hline f(n_8) & f(n_5) & f(n_3) \\ \hline\end{array} \rightarrow \begin{array}{|c|c|c|}\hline f(n_6) & f(n_4) & f(n_1) \\ \hline f(n_7) & s' & f(n_2) \\ \hline f(n_8) & f(n_5) & f(n_3) \\ \hline\end{array}$$

where $f(a) = a$ for all $a \in \Gamma$ and $f(\vdash) = \top, f(\top) = \dashv, f(\dashv) = \bot, f(\bot) = \vdash, f(\#) = \#$.

The mirroring of the rules can be done similarly. □

Observation 2. *The class* $\mathcal{L}(\mathsf{2LCRA})$ *is closed under binary intersection.*

Proof. Given two automata $\mathcal{M}_1, \mathcal{M}_2$ we can construct a 2LCRA $\mathcal{M}$ that simulates both of them by first replacing each input symbol s by a pair (s, s) and then performing computations of $\mathcal{M}_1$ and $\mathcal{M}_2$ on the first and the second element of the pairs, respectively. $\mathcal{M}$'s weight function is $\delta(a) = 1 + \delta_1(a) + \delta_2(a)$ for input symbols and $\delta((a,b)) = \delta_1(a) + \delta_2(b)$ for the pairs. This way a symbol's weight is zero only if the symbol is a pair of zero symbols (w.r.t. δ_1 and δ_2) and the picture is only accepted if both computations terminated successfully. $\square$

Proposition 2. *The class* $\mathcal{L}(\mathsf{2LCRA})$ *is closed under projection.*

Proof. Let $\mathcal{M} = (\Sigma, \Gamma, \delta, \mu)$ be a $\mathsf{2LCRA}$ and $\pi : \Sigma \to \Sigma'$ be a projection. We w.l.o.g. assume that $\Gamma \cap \Sigma' = \emptyset$. To accept $\pi(L(\mathcal{M}))$ we can construct an automaton $\mathcal{M}_\pi = (\Sigma', \Gamma', \delta' \mu')$ where $\Gamma' = \Gamma \cup \Sigma'$ and δ' contains all the rules from δ as well as new rules $\boxed{a'} \to \boxed{a}$ for all $a \in \Sigma, a' \in \Sigma'$ such that $\pi(a) = a'$. The weight function μ' is $\mu' = \mu(a)$ for $a \in \Gamma$ and $\mu'(a) = 1 + \max_{s \in \Gamma}(\mu(s))$ for $a \in \Sigma'$. $\square$

Proposition 3. *The class* $\mathcal{L}(\mathsf{2LCRA})$ *is closed under binary union.*

Proof. Let $\mathcal{M}_1 = (\Sigma, \Gamma_1, \delta_1, \mu_1)$ and let $\mathcal{M}_2 = (\Sigma, \Gamma_2, \delta_2, \mu_2)$ be two $\mathsf{2LCRA}$. W.l.o.g. we assume that $\Gamma_1 \cap \Gamma_2 = \Sigma$ and also that no symbol of the input alphabet has zero weight with respect to either μ_1 or μ_2.

For $\mathcal{M}_1$ we can trivially create an equivalent automaton $\mathcal{M}'_1 = (\Sigma_1, \Gamma'_1, \delta'_1, \mu'_1)$ operating on a renaming of Σ (i.e. $\Sigma_1 \cap \Sigma = \emptyset$ and there exists a bijection $\pi_1 : \Sigma_1 \to \Sigma$ such that $\mu'_1(s) = \mu_1(\pi_1(s))$ for all $s \in \Sigma_1$ and $\pi_1(L(\mathcal{M}'_1)) = L(\mathcal{M}_1)$. In the same way we can construct $\mathcal{M}'_2 = (\Sigma_2, \Gamma'_2, \delta'_2, \mu'_2)$ such that $\Gamma'_1 \cap \Gamma'_2 = \emptyset$.

Now we can finally build an automaton accepting $L(\mathcal{M}_1) \cup L(\mathcal{M}_2) = \pi_1(L(\mathcal{M}'_1)) \cup \pi_2(L(\mathcal{M}'_2))$. It is a tuple $\mathcal{M} = (\Sigma, \Gamma, \delta, \mu)$ where

$\Gamma = \Sigma \cup \Gamma'_1 \cup \Gamma'_2$, the weight function is

- $\mu(a) = \mu'_1(a)$ for $a \in \Gamma'_1$,
- $\mu(a) = \mu'_2(a)$ for $a \in \Gamma'_2$ and
- $\mu(a) = \mu'_1(\pi_1^{-1}(a)) + \mu'_2(\pi_2^{-1}(a))$ for $a \in \Sigma$,

and the rewriting rules are $\delta = \delta'_1 \cup \delta'_2 \cup \delta_\pi$ where δ_π contains the projection rules $\boxed{a} \to \boxed{\pi_1^{-1}(a)}$ and $\boxed{a} \to \boxed{\pi_2^{-1}(a)}$ for all $a \in \Sigma$.

In a computation of $\mathcal{M}$, every cell is first rewritten to a symbol from either Σ_1 or Σ_2. Then it must be rewritten at least once by a rule from $\mathcal{M}'_1$ or $\mathcal{M}'_2$ to reach zero weight. If a part of the picture is rewritten to symbols from Σ_1 and an another part is rewritten symbols from Σ_2, the cells on the interface between these parts cannot be rewritten further and the picture cannot be accepted. Therefore the only accepting computations are equivalent to renaming the whole picture to one alphabet and subsequent accepting by the respective automaton (although not necessarily in this strict order — the automaton can begin rewriting the renamed parts before the picture is completely renamed). $\square$

Proposition 4. *The class* $\mathcal{L}(\mathsf{2LCRA})$*is closed under binary horizontal and vertical concatenation.*

Proof. An automaton accepting a concatenation of two languages can be constructed similarly as with the union. For horizontal concatenation, this automaton can rename left part of a picture to an alphabet of the first automaton and the right part to an alphabet of the other one and then perform two independent computations over the separated parts. The left computation treats the right side symbols as right sentinels an vice versa. In a successful computation, there obviously cannot be any interface between the two alphabets except for the single vertical border corresponding to the concatenation point. ☐

Lemma 2. *The class $\mathcal{L}(\mathsf{2LCRA})$ is not closed under complement.*

Proof. As with the similar proof for 2RTA in [2], we will use the language of pictures consisting of two identical vertically concatenated squares, with a slight modification.

$$L = \{s \ominus l \ominus s | l \in \Sigma^{1,n}, s \in \Sigma^{n,n}, n \in \mathbb{N}\}, \Sigma = \{\mathbf{a}, \mathbf{b}\}$$

The complement of this language can be recognized by a 2LCRA that either verifies that a picture P is not of a size $(2n+1, n)$ or checks that the two squares are not equal.

The former procedure is done by attempting to send a signal from the position $(1, 1)$ to the position $(1, 2\text{cols}(P) + 1)$ (see Fig. 4.1a). If this signal ends up somewhere else than in the bottom left corner, the picture is rewritten to zero and accepted. The signal is transmitted using the rules

$$\begin{array}{|c|c|}\hline \# & \top \\ \hline \vdash & s \\ \hline\end{array} \to \begin{array}{|c|c|}\hline \# & \top \\ \hline \vdash & \searrow \\ \hline\end{array}, \quad \begin{array}{|c|c|}\hline \searrow & s \\ \hline t & u \\ \hline\end{array} \to \begin{array}{|c|c|}\hline \searrow & s \\ \hline t & \searrow \\ \hline\end{array}, \quad \begin{array}{|c|c|}\hline \searrow & \dashv \\ \hline s & \dashv \\ \hline\end{array} \to \begin{array}{|c|c|}\hline \searrow & \dashv \\ \hline \circlearrowright & \dashv \\ \hline\end{array},$$

$$\begin{array}{|c|c|}\hline \circlearrowright & \dashv \\ \hline s & \dashv \\ \hline\end{array} \to \begin{array}{|c|c|}\hline \circlearrowright & \dashv \\ \hline \swarrow & \dashv \\ \hline\end{array}, \quad \begin{array}{|c|c|}\hline s & \swarrow \\ \hline t & u \\ \hline\end{array} \to \begin{array}{|c|c|}\hline s & \swarrow \\ \hline \swarrow & u \\ \hline\end{array}$$

for $s, t, u \in \{\mathbf{a}, \mathbf{b}\}$. The fact that the signal had not arrived in the correct corner is acknowledged by the rules

$$\begin{array}{|c|c|}\hline \searrow & \dashv \\ \hline \bot & \# \\ \hline\end{array} \to \begin{array}{|c|c|}\hline \mathbf{0} & \dashv \\ \hline \bot & \# \\ \hline\end{array}, \quad \begin{array}{|c|c|}\hline \circlearrowright & \dashv \\ \hline \bot & \# \\ \hline\end{array} \to \begin{array}{|c|c|}\hline \mathbf{0} & \dashv \\ \hline \bot & \# \\ \hline\end{array},$$

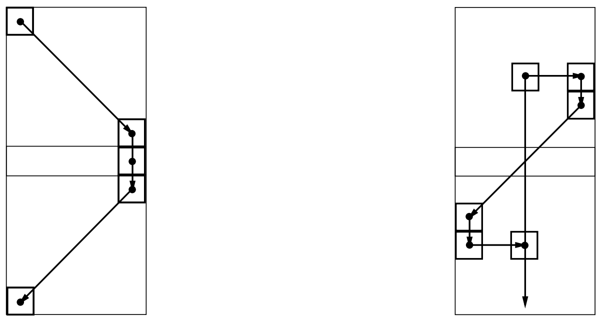
$$\begin{array}{|c|c|}\hline s & \swarrow \\ \hline \bot & \bot \\ \hline\end{array} \to \begin{array}{|c|c|}\hline s & \mathbf{0} \\ \hline \bot & \bot \\ \hline\end{array}, \quad \begin{array}{|c|c|}\hline \vdash & \swarrow \\ \hline \vdash & s \\ \hline\end{array} \to \begin{array}{|c|c|}\hline \vdash & \mathbf{0} \\ \hline \vdash & s \\ \hline\end{array},$$

where $\mathbf{s} \in \{\mathbf{a}, \mathbf{b}\}$ and $\mathbf{0}$ is a zero-weight symbol, which is, after its first appearance, spread to the whole picture.

The latter procedure uses auxiliary symbols disjoint with those used above. It begins by nondeterministically choosing a field (i, j) in the upper square that is supposedly not equal to the corresponding field in the lower square. This field is marked by either the rule $\boxed{a} \to \boxed{a_0}$ or the rule $\boxed{b} \to \boxed{b_0}$. After that, the rest of the picture is frozen (by rewriting every $\mathbf{a}$ to $\mathbf{a}'$ and every $\mathbf{b}$ to $\mathbf{b}'$) so that no other field is selected (using, for example, a technique similar to the one described in 3.5). The information about the contents of the selected field is then transferred to the field $(i + \text{cols}(P) + 1, j)$ (see Fig. 4.1b). First, the value of the marked field is announced to the column below it using the rules

$$\begin{array}{|c|}\hline \mathbf{a_0}, \mathbf{a_a}, \mathbf{b_a} \\ \hline \mathbf{a}' \\ \hline\end{array} \to \begin{array}{|c|}\hline \mathbf{a_0}, \mathbf{a_a}, \mathbf{b_a} \\ \hline \mathbf{a_a} \\ \hline\end{array}, \quad \begin{array}{|c|}\hline \mathbf{b_0}, \mathbf{a_b}, \mathbf{b_b} \\ \hline \mathbf{a}' \\ \hline\end{array} \to \begin{array}{|c|}\hline \mathbf{b_0}, \mathbf{a_b}, \mathbf{b_b} \\ \hline \mathbf{a_b} \\ \hline\end{array},$$

$$\begin{array}{|c|}\hline \mathbf{a_0}, \mathbf{a_a}, \mathbf{b_a} \\ \hline \mathbf{b}' \\ \hline\end{array} \to \begin{array}{|c|}\hline \mathbf{a_0}, \mathbf{a_a}, \mathbf{b_a} \\ \hline \mathbf{b_a} \\ \hline\end{array}, \quad \begin{array}{|c|}\hline \mathbf{b_0}, \mathbf{a_b}, \mathbf{b_b} \\ \hline \mathbf{b}' \\ \hline\end{array} \to \begin{array}{|c|}\hline \mathbf{b_0}, \mathbf{a_b}, \mathbf{b_b} \\ \hline \mathbf{b_b} \\ \hline\end{array}.$$



(a) Sending a signal from the top left corner to the bottom left corner.

(b) Sending information between corresponding fields of two squares.

Figure 4.1: Techniques used for recognizing a complement of the language of two identical, vertically concatenated squares

Then, a signal is sent to ascertain the correct row using the rules

$$\begin{array}{|c|}\hline \mathbf{a_0}, \mathbf{b_0} \\ \hline\end{array} \to \begin{array}{|c|}\hline \rightarrow \\ \hline\end{array}, \quad \begin{array}{|c|c|}\hline \rightarrow & s' \\ \hline\end{array} \to \begin{array}{|c|c|}\hline \rightarrow & \rightarrow \\ \hline\end{array},$$

$$\begin{array}{|c|c|}\hline \rightarrow & \dashv \\ \hline s' & \dashv \\ \hline\end{array} \to \begin{array}{|c|c|}\hline \rightarrow & \dashv \\ \hline \swarrow & \dashv \\ \hline\end{array}, \quad \begin{array}{|c|c|}\hline s' & \swarrow \\ \hline t' & u' \\ \hline\end{array} \to \begin{array}{|c|c|}\hline s' & \swarrow \\ \hline \swarrow & u' \\ \hline\end{array}, \quad \begin{array}{|c|c|}\hline \vdash & \swarrow \\ \hline \vdash & s' \\ \hline\end{array} \to \begin{array}{|c|c|}\hline \vdash & \swarrow \\ \hline \vdash & \rightarrow \\ \hline\end{array}$$

where $s', t', u' \in \{\mathbf{a}', \mathbf{b}', \mathbf{a_a}, \mathbf{b_a}, \mathbf{a_b}, \mathbf{b_b}\}$. This signal, in its final phase, eventually crosses the j-th column in the row $(i + \mathrm{cols}(P) + 1)$ and we can check whether the two corresponding fields differ using rules

$$\begin{array}{|c|c|}\hline \rightarrow & \mathbf{a_b}, \mathbf{b_a} \\ \hline\end{array} \to \begin{array}{|c|c|}\hline \rightarrow & \mathbf{0} \\ \hline\end{array}, \quad \begin{array}{|c|c|}\hline \vdash & \swarrow \\ \hline \vdash & \mathbf{a_b}, \mathbf{b_a} \\ \hline\end{array} \to \begin{array}{|c|c|}\hline \vdash & \swarrow \\ \hline \vdash & \mathbf{0} \\ \hline\end{array}.$$

Again, $\mathbf{0}$ is a zero weight symbol that can be propagated to the whole picture.

Now we will prove that $L \notin \mathcal{L}(\mathsf{2LCRA})$ by contradiction. Let us assume that there exists a restarting automaton $\mathcal{M} = (\Sigma, \Gamma, \delta, \mu)$ that recognizes L. For every picture in L we take one accepting computation and informally define its *signature* as the complete information about rewritings of symbols on the picture's horizontal center line (l). Each signature is a set of rewritings (identified by a position and the rule used) ordered w.r.t. their order in the whole computation.

Now we proceed to count the maximum number of distinct signatures of computations over pictures of fixed width n. At one position, each rule can be used at most once, which gives us at most $2^{|\delta|}$ possible combinations of used rules. For all positions on the line combined, that sums up to $(2^{|\delta|})^n$ combinations. Each combination contains at most $n|\delta|$ rewritings, which can be ordered in $(n|\delta|)!$ ways. Therefore the upper bound for the number of distinct signatures is

$$(2^{|\delta|})^n (n|\delta|)!.$$

On the other hand, there are $2^{(n^2)}$ squares of size n over the alphabet $\{\mathbf{a}, \mathbf{b}\}$. The limit

$$\lim_{n\to\infty} \frac{2^{(n^2)}}{(2^{|\delta|})^n (n|\delta|)!} \geq \lim_{n\to\infty} \frac{2^{(n^2)}}{(2^{|\delta|}|\delta|n))^{(n|\delta|)}} = \infty$$

dictates that for a large enough n there are more squares than signatures of computations over pictures made up from them. So there must exist two different squares s_1, s_2 such that there are accepting computations of $\mathcal{M}$ over $s_1 \ominus l \ominus s_1$ and $s_2 \ominus l \ominus s_2$ sharing the same signature.

We will show that $\mathcal{M}$ accepts the picture $s_1 \ominus l \ominus s_2 \notin L$. An accepting computation here performs rewritings from the signature as well as those that were done in the upper half of $s_1 \ominus l \ominus s_1$ and the lower half of $s_2 \ominus l \ominus s_2$ during their respective accepting computations. All of these rewritings are done in order that respects the partial ordering imposed by the two original computations over $s_1 \ominus l \ominus s_1$ and $s_2 \ominus l \ominus s_2$. Specifically, if a rewriting in the upper half was performed between two rewritings on the center line during the computation on $s_1 \ominus l \ominus s_1$, it must also be performed between them in the computation on $s_1 \ominus l \ominus s_2$. An analogous principle is used for ordering the rewritings performed in the lower square during the computation on $s_2 \ominus l \ominus s_2$.

That way, every time a rewriting takes place in the upper (or lower) square, all of the fields in the neighboring positions (from the square and the center line) are the same as in the original computation, and therefore the rewriting is possible. The same is true for the rewritings on the center line, because at the time we perform such rewritings, the current picture is, both below and above the line, in a state that allows the usage of the particular rule.

$\mathcal{M}$ accepting this picture contradicts the assumption that $\mathcal{M}$ recognizes L. Therefore such automaton cannot exist.

□

4.2 Rule format

We have defined the automaton to use rules in the form of a 3-by-3 sub-picture, where the central symbol is rewritten. This expresses, in the most general form, that the rewriting of a cell is based on its immediate neighborhood. There are, however, other formats of rules that yield automata of equal power. Here we will explore the most notable of them.

4.2.1 Tile format

This is the format used by 2RTA. Each rule describes rewriting of a tile

$$\begin{array}{|c|c|}\hline s_1 & s_2 \\ \hline s_3 & s_4 \\ \hline\end{array} \to \begin{array}{|c|c|}\hline s_1' & s_2' \\ \hline s_3' & s_4' \\ \hline\end{array}$$

where for exactly one $i \in \{1,2,3,4\}$, $s_i \neq s_i'$.

With such rules, any rule of the form

$$r = \begin{array}{|c|c|c|}\hline n_1 & n_2 & n_3 \\ \hline n_4 & s & n_5 \\ \hline n_6 & n_7 & n_8 \\ \hline\end{array} \to \begin{array}{|c|c|c|}\hline n_1 & n_2 & n_3 \\ \hline n_4 & s' & n_5 \\ \hline n_6 & n_7 & n_8 \\ \hline\end{array}$$

can be simulated by the set of following four rules:

$$\begin{array}{|c|c|}\hline n_1 & n_2 \\ \hline n_4 & s \\ \hline \end{array} \rightarrow \begin{array}{|c|c|}\hline n_1 & n_2 \\ \hline n_4 & s_r^1 \\ \hline \end{array}$$

$$\begin{array}{|c|c|}\hline n_2 & n_3 \\ \hline s_r^1 & n_5 \\ \hline \end{array} \rightarrow \begin{array}{|c|c|}\hline n_2 & n_3 \\ \hline s_r^2 & n_5 \\ \hline \end{array}$$

$$\begin{array}{|c|c|}\hline n_4 & s_r^2 \\ \hline n_6 & n_7 \\ \hline \end{array} \rightarrow \begin{array}{|c|c|}\hline n_4 & s_r^3 \\ \hline n_6 & n_7 \\ \hline \end{array}$$

$$\begin{array}{|c|c|}\hline s_r^3 & n_5 \\ \hline n_7 & n_8 \\ \hline \end{array} \rightarrow \begin{array}{|c|c|}\hline s' & n_5 \\ \hline n_7 & n_8 \\ \hline \end{array}$$

where s_r^1, s_r^2, s_r^3 are new symbols unique to the rule r, with weights such that $\mu(s) > \mu(s_r^1) > \mu(s_r^2) > \mu(s_r^3) > \mu(s')$ (to accommodate these weights, the weights of the original alphabet can be multiplied by four to ensure that $\mu(s)-\mu(s') \geq 4$).

During the rewriting of a cell by this sequence, no neighbor can be successfully rewritten to a symbol of the original working alphabet. Thus these sequences are practically atomic. Therefore, using these rules instead of the original rules does not expand the accepted language.

4.2.2 Cross format

Here we will show that for every 2LCRA $\mathcal{M} = (\Sigma, \Gamma, \delta, \mu)$ there exists an automaton $\mathcal{M}'$ accepting the same language, which utilizes rules in the following form:

$$\begin{array}{ccc}\cline{2-2} & \multicolumn{1}{|c|}{n_\mathrm{u}} & \\ \hline \multicolumn{1}{|c|}{n_\mathrm{l}} & \multicolumn{1}{c|}{s} & \multicolumn{1}{c|}{n_\mathrm{r}} \\ \hline & \multicolumn{1}{|c|}{n_\mathrm{d}} & \\ \cline{2-2} \end{array} \rightarrow \begin{array}{ccc}\cline{2-2} & \multicolumn{1}{|c|}{n_\mathrm{u}} & \\ \hline \multicolumn{1}{|c|}{n_\mathrm{l}} & \multicolumn{1}{c|}{s'} & \multicolumn{1}{c|}{n_\mathrm{r}} \\ \hline & \multicolumn{1}{|c|}{n_\mathrm{d}} & \\ \cline{2-2} \end{array}$$

To do that, we will describe an automaton $\mathcal{M}' = (\Sigma, \Gamma', \delta', \mu')$ that, in order to simulate $\mathcal{M}$, keeps at every position in the picture information about its four neighbors.

The working alphabet Γ' contains the input alphabet Σ, a special zero-weight symbol $\mathbf{0}$ and all six-tuples

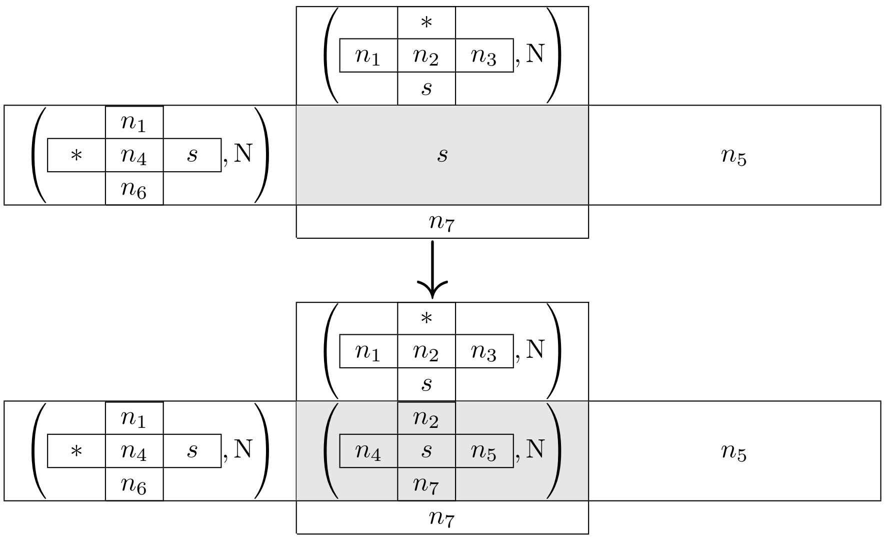
$$(c, u, l, r, d, x) \in \Gamma_e \times \Gamma_e \times \Gamma_e \times \Gamma_e \times \Gamma_e \times \{\mathrm{N}, \mathrm{U}, \mathrm{L}, \mathrm{R}, \mathrm{D}\}, \Gamma_e = \Gamma \cup \mathcal{S}$$

or, graphically,

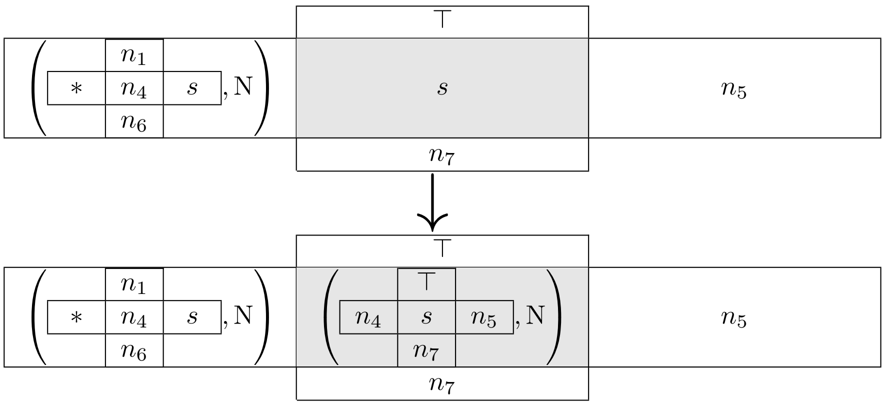
$$\left(\begin{array}{ccc}\cline{2-2} & \multicolumn{1}{|c|}{u} & \\ \hline \multicolumn{1}{|c|}{l} & \multicolumn{1}{c|}{c} & \multicolumn{1}{c|}{r} \\ \hline & \multicolumn{1}{|c|}{d} & \\ \cline{2-2} \end{array}, x \right).$$

These symbols describe contents of a cell and its neighbors and a lock that will be instrumental in keeping this structure consistent.

The computation of $\mathcal{M}'$ begins by informing cells of their neighbors using rules

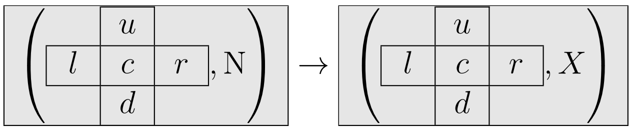


(where each $*$ stands for any symbol) and their apparent variations for positions at the edges of the picture, for example



Then, the steps of $\mathcal{M}$ are simulated. Simulation of one rewriting consists of locking the four neighbors of the rewritten position, performing the rewriting and finally updating and unlocking the neighbors.

To lock a cell means rewriting the last element of the tuple by the rules

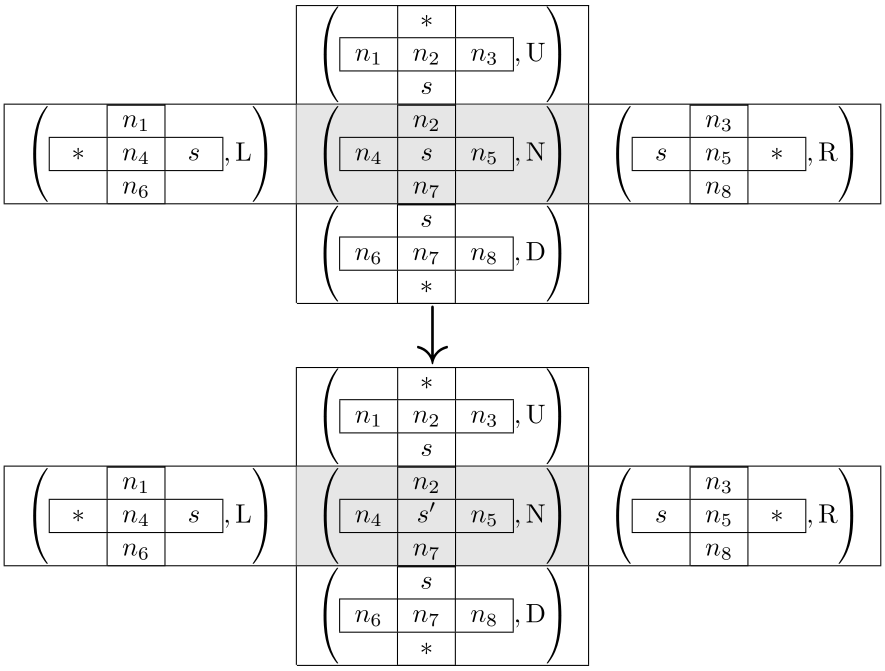


where $X \in \{\mathrm{U}, \mathrm{L}, \mathrm{R}, \mathrm{D}\}$.

The actual rewriting corresponding to rule

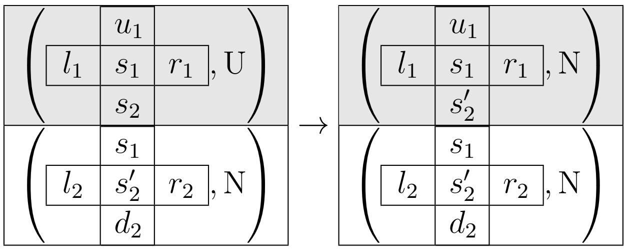
$$\begin{array}{|c|c|c|}\hline n_1 & n_2 & n_3 \\ \hline n_4 & s & n_5 \\ \hline n_6 & n_7 & n_8 \\ \hline \end{array} \rightarrow \begin{array}{|c|c|c|}\hline n_1 & n_2 & n_3 \\ \hline n_4 & s' & n_5 \\ \hline n_6 & n_7 & n_8 \\ \hline \end{array}$$

is carried out by a rule matching



or a variation for borders of the picture.

Then, the neighbors are updated and unlocked by rules



and their analogues for left (L), right (R) and down (D) locks.

For these rules to form a valid automaton, the tuples must be weighed with respect to their lexicographical ordering (with locks sorted for example D < R < L < U < N)

At the end of the simulation, the tuples where the center symbol has a weight of zero according to μ are rewritten to the special 0 symbol.

Observe that by using a similar principle for of keeping the states of neighbors in every field, we can simulate rules matching 5-by-5 sub-pictures with 3-by-3 rules.

4.2.3 Domino format

Perhaps the simplest format of the rules is that which only uses one of the four closest neighbors of the cell and the cell itself:

$$\begin{array}{|c|}\hline n\\\hline s\\\hline\end{array}\rightarrow\begin{array}{|c|}\hline n\\\hline s'\\\hline\end{array},\quad \begin{array}{|c|}\hline s\\\hline n\\\hline\end{array}\rightarrow\begin{array}{|c|}\hline s'\\\hline n\\\hline\end{array},\quad \begin{array}{|c|c|}\hline n & s\\\hline\end{array}\rightarrow\begin{array}{|c|c|}\hline n & s'\\\hline\end{array},\quad \begin{array}{|c|c|}\hline s & n\\\hline\end{array}\rightarrow\begin{array}{|c|c|}\hline s' & n\\\hline\end{array}$$

With such, a cross rule

$$r = \begin{array}{|c|c|c|}\cline{2-2}\multicolumn{1}{c|}{} & n_{\mathrm{u}} & \multicolumn{1}{c}{}\\\hline n_{\mathrm{l}} & s & n_{\mathrm{r}}\\\hline\multicolumn{1}{c|}{} & n_{\mathrm{d}} & \multicolumn{1}{c}{}\\\cline{2-2}\end{array}\rightarrow\begin{array}{|c|c|c|}\cline{2-2}\multicolumn{1}{c|}{} & n_{\mathrm{u}} & \multicolumn{1}{c}{}\\\hline n_{\mathrm{l}} & s' & n_{\mathrm{r}}\\\hline\multicolumn{1}{c|}{} & n_{\mathrm{d}} & \multicolumn{1}{c}{}\\\cline{2-2}\end{array}$$

(where $s, s' \in \Gamma$, $n_{\mathrm{u}}, n_{\mathrm{l}}, n_{\mathrm{r}}, n_{\mathrm{d}} \in \Gamma \cup \mathcal{S}$ for the working alphabet Γ) can be, like in the case of conversion between tile and 3-by-3 format, simulated via

$$\begin{array}{|c|}\hline n_{\mathrm{u}} \\ \hline s \\ \hline\end{array} \rightarrow \begin{array}{|c|}\hline n_{\mathrm{u}} \\ \hline s_r^1 \\ \hline\end{array}, \quad \begin{array}{|c|}\hline s_r^1 \\ \hline n_{\mathrm{d}} \\ \hline\end{array} \rightarrow \begin{array}{|c|}\hline s_r^2 \\ \hline n_{\mathrm{d}} \\ \hline\end{array}, \quad \begin{array}{|c|c|}\hline n_{\mathrm{l}} & s_r^2 \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline n_{\mathrm{l}} & s_r^3 \\ \hline\end{array}, \quad \begin{array}{|c|c|}\hline s_r^3 & n_{\mathrm{l}} \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline s' & n_{\mathrm{l}} \\ \hline\end{array}$$

and therefore the automata utilizing this format of rules still retain the power of the originally defined 2LCRA.

4.3 Comparison with other models

In this section we insert our model into the hierarchy of existing models. The most important observation here is that 2LCRA are equally powerful as sgraffito automata. This fact immediately gives us comparisons with various other models.

4.3.1 Recognizable picture languages

Observation 3. $L \in \mathsf{REC} \implies L \cup \{\Lambda\} \in \mathcal{L}(\mathsf{2LCRA})$.

Proof. Let $\mathcal{T} = (\Sigma, \Gamma, \Theta, \pi)$ be a tiling system such that $\Sigma \cap \Gamma = \emptyset$.

Then we can build a 2LCRA that recognizes pictures from $L(\mathcal{T})$ by rewriting it in one pass using rules

- for fields not on the right or the bottom edge of the picture:

$$\begin{array}{|c|c|c|}\hline \gamma_1 & \gamma_2 & \sigma_3 \\ \hline \gamma_4 & \pi(\gamma) & \sigma_5 \\ \hline \sigma_6 & \sigma_7 & \sigma_8 \\ \hline\end{array} \rightarrow \begin{array}{|c|c|c|}\hline \gamma_1 & \gamma_2 & \sigma_3 \\ \hline \gamma_4 & \gamma & \sigma_5 \\ \hline \sigma_6 & \sigma_7 & \sigma_8 \\ \hline\end{array}$$

  for all $\gamma \in \Gamma, \gamma_1, \gamma_2, \gamma_4 \in (\Gamma \cup \mathcal{S}), \sigma_3, \sigma_6 \in (\Sigma \cup \mathcal{S}), \sigma_5, \sigma_7, \sigma_8 \in \Sigma$ such that $\begin{array}{|c|c|}\hline \gamma_1 & \gamma_2 \\ \hline \gamma_4 & \gamma \\ \hline\end{array} \in \Theta$,

- for fields on the right edge of the picture, but not in the bottom right corner:

$$\begin{array}{|c|c|c|}\hline \gamma_1 & \gamma_2 & s_3 \\ \hline \gamma_4 & \pi(\gamma) & \dashv \\ \hline \sigma_6 & \sigma_7 & \dashv \\ \hline\end{array} \rightarrow \begin{array}{|c|c|c|}\hline \gamma_1 & \gamma_2 & s_3 \\ \hline \gamma_4 & \gamma & \dashv \\ \hline \sigma_6 & \sigma_7 & \dashv \\ \hline\end{array}$$

  for all $\gamma \in \Gamma, \gamma_1, \gamma_2, \gamma_4 \in (\Gamma \cup \mathcal{S}), \sigma_6 \in (\Sigma \cup \mathcal{S}), \sigma_7, \in \Sigma, s_3 \in \mathcal{S}$ such that $\begin{array}{|c|c|}\hline \gamma_1 & \gamma_2 \\ \hline \gamma_4 & \gamma \\ \hline\end{array}, \begin{array}{|c|c|}\hline \gamma_2 & s_3 \\ \hline \gamma & \dashv \\ \hline\end{array} \in \Theta$,

- for fields on the bottom edge of the picture, but not in the bottom right corner:

$$\begin{array}{|c|c|c|}\hline \gamma_1 & \gamma_2 & \sigma_3 \\ \hline \gamma_4 & \pi(\gamma) & \sigma_5 \\ \hline s_6 & \perp & \perp \\ \hline\end{array} \rightarrow \begin{array}{|c|c|c|}\hline \gamma_1 & \gamma_2 & \sigma_3 \\ \hline \gamma_4 & \gamma & \sigma_5 \\ \hline s_6 & \perp & \perp \\ \hline\end{array}$$

  for all $\gamma \in \Gamma, \gamma_1, \gamma_2, \gamma_4 \in (\Gamma \cup \mathcal{S}), \sigma_3 \in (\Sigma \cup \mathcal{S}), \sigma_5 \in \Sigma, s_6 \in \mathcal{S}$ such that $\begin{array}{|c|c|}\hline \gamma_1 & \gamma_2 \\ \hline \gamma_4 & \gamma \\ \hline\end{array}, \begin{array}{|c|c|}\hline \gamma_4 & \gamma \\ \hline s_6 & \perp \\ \hline\end{array} \in \Theta$

- and for a field in the bottom right corner of the picture:

$$\begin{array}{|c|c|c|}\hline \gamma_1 & \gamma_2 & s_3 \\ \hline \gamma_4 & \pi(\gamma) & \dashv \\ \hline s_6 & \perp & \# \\ \hline \end{array} \to \begin{array}{|c|c|c|}\hline \gamma_1 & \gamma_2 & s_3 \\ \hline \gamma_4 & \gamma & \dashv \\ \hline s_6 & \perp & \# \\ \hline \end{array}$$

for all $\gamma \in \Gamma, \gamma_1, \gamma_2, \gamma_4 \in (\Gamma \cup \mathcal{S}), s_3, s_6, \in \mathcal{S}$ such that

$\begin{array}{|c|c|}\hline \gamma_1 & \gamma_2 \\ \hline \gamma_4 & \gamma \\ \hline \end{array}, \begin{array}{|c|c|}\hline \gamma_2 & s_3 \\ \hline \gamma & \dashv \\ \hline \end{array}, \begin{array}{|c|c|}\hline \gamma_4 & \gamma \\ \hline s_6 & \perp \\ \hline \end{array}, \begin{array}{|c|c|}\hline \gamma & \dashv \\ \hline \perp & \# \\ \hline \end{array} \in \Theta.$

The weight function μ is defined as $\mu(\sigma) = 1$ for all $\sigma \in \Sigma$ and $\mu(\gamma) = 0$ for all $\gamma \in \Gamma$. □

4.3.2 Two-dimensional sgraffito automaton

Theorem 7. *For every picture language L, $L \in \mathcal{L}(\mathsf{2SA}) \iff L \cup \{\Lambda\} \in \mathcal{L}(\mathsf{2LCRA})$.*

To verify this claim we will show how a limited context restarting automaton can be simulated by a sgraffito automaton and vice versa.

Lemma 3. $\mathcal{L}(\mathsf{2LCRA}) \subseteq \mathcal{L}(\mathsf{SA})$.

Proof. Let $\mathcal{M} = (\Sigma, \Gamma, \delta, \mu)$ be an arbitrary 2LCRA. To prove the inclusion we will construct a sgraffito automaton $\mathcal{M}_\mathrm{S}$ accepting the language $L(\mathcal{M}_\mathrm{S}) = L(\mathcal{M})$. We will w.l.o.g. assume that $\mathcal{M}$ uses rules in the cross format and that all its input symbols have non-zero weight.

For a rule

$$r = \begin{array}{ccc} \cline{2-2} & \multicolumn{1}{|c|}{n_\mathrm{u}} & \\ \hline \multicolumn{1}{|c|}{n_\mathrm{l}} & \multicolumn{1}{c|}{s} & \multicolumn{1}{c|}{n_\mathrm{r}} \\ \hline & \multicolumn{1}{|c|}{n_\mathrm{d}} & \\ \cline{2-2} \end{array} \to \begin{array}{ccc} \cline{2-2} & \multicolumn{1}{|c|}{n_\mathrm{u}} & \\ \hline \multicolumn{1}{|c|}{n_\mathrm{l}} & \multicolumn{1}{c|}{s'} & \multicolumn{1}{c|}{n_\mathrm{r}} \\ \hline & \multicolumn{1}{|c|}{n_\mathrm{d}} & \\ \cline{2-2} \end{array}$$

we denote the left neighbor n_l by $\mathrm{ln}(r)$, the right neighbor n_r by $\mathrm{rn}(r)$, the upper neighbor n_u by $\mathrm{un}(r)$, the lower neighbor n_d by $\mathrm{dn}(r)$, the central symbol s by $\mathrm{ctr}(r)$ and the result s' by $\mathrm{res}(r)$.

Definition 14. *Let $\mathcal{M} = (\Sigma, \Gamma, \delta, \mu)$ be a 2LCRA and $s \in \Sigma$ an input symbol. A* reduction of s by $\mathcal{M}$ *is a sequence of rewriting rules $(r_1, r_2, \ldots, r_n), r_i \in \delta$ such that*

1. $\mathrm{ctr}(r_1) = s$
2. $\mu(\mathrm{res}(r_n)) = 0$
3. $(\forall i \in \{2, \ldots, n\}) : \mathrm{res}(r_{i-1}) = \mathrm{ctr}(r_i)$.

We define $R(\mathcal{M})$ as the set of all reductions of all symbols in Σ by $\mathcal{M}$.

In other words, a *reduction* is a way in which one particular symbol can be rewritten to a zero weighed result.

Definition 15. *Let $\mathcal{M} = (\Sigma, \Gamma, \delta, \mu)$ be a* 2LCRA *and let $r = (r_1, \ldots, r_m)$ be a reduction of $a \in \Sigma$ by $\mathcal{M}$ and $s = (s_1, \ldots, s_n)$ be a reduction of $b \in \Sigma$ by $\mathcal{M}$. We say that r is* horizontally compatible with *s if there exists a linear ordering $\leq_c$ of $\{(r_1, \leftarrow), \ldots, (r_m, \leftarrow), (s_1, \rightarrow), \ldots, (s_n, \rightarrow)\}$ such that for all $i \in \{1, \ldots, m\}$, $j \in \{1, \ldots, n\}$*

1. $(r_i, \leftarrow) <_c (r_{i+1}, \leftarrow)$ *if* $i < m$,
   $(s_j, \rightarrow) <_c (s_{j+1}, \rightarrow)$ *if* $j < n$,
2. $(r_i, \leftarrow) <_c (s_1, \rightarrow) \implies (\mathrm{rn}(r_i) = b)$,
   $(s_j, \rightarrow) <_c (r_1, \leftarrow) \implies (\mathrm{ln}(s_j) = a)$,
3. $(r_i, \leftarrow) >_c (s_n, \rightarrow) \implies (\mathrm{rn}(r_i) = \mathrm{res}(s_n))$,
   $(s_j, \rightarrow) >_c (r_m, \leftarrow) \implies (\mathrm{ln}(s_j) = \mathrm{res}(r_m))$,
4. $(s_j, \rightarrow) <_c (r_i, \leftarrow) < (s_{j+1}, \rightarrow) \implies (\mathrm{rn}(r_i) = \mathrm{res}(s_j))$ *if* $j < n$,
   $(r_i, \leftarrow) <_c (s_j, \rightarrow) < (r_{i+1}, \leftarrow) \implies (\mathrm{ln}(s_j) = \mathrm{res}(r_i))$ *if* $i < m$.

This defines a relation C_h on the set of all reductions of all symbols in Σ:

$$rC_\mathrm{h}s \iff r \text{ is horizontally compatible with } s.$$

We can see that horizontal compatibility is a necessary (though not sufficient) condition for two reductions happening in the neighboring positions in the same row. We can similarly define a relation of *vertical compatibility* (C_v) for neighbors in the same column. Note that these relations are not symmetrical.

Observation 4. *For a pair of reductions r, s there exists at most one ordering $\leq_c$ witnessing their horizontal compatibility and at most one ordering $\leq'_c$ witnessing their vertical compatibility.*

With these definitions, we can now describe a way of recognizing pictures in $\mathcal{L}(\mathcal{M})$ that is achievable by a sgraffito automaton. The recognition consists of guessing the correct reduction for every position in the picture and verifying whether these reductions can be assembled into a valid computation.

Lemma 4. *Let $\mathcal{M} = (\Sigma, \Gamma, \delta, \mu)$ be a* 2LCRA *and $P \in \Sigma^{m,n}$ be a picture. Then $P \in \mathcal{L}(\mathcal{M})$ if and only if there exists a mapping $f : \{1, \ldots, m\} \times \{1, \ldots, n\} \rightarrow R(\mathcal{M})$ between positions in P an reductions by $\mathcal{M}$ such that for all $i \in \{1, \ldots, m\}$, $j \in \{1, \ldots, n\}$*

1. $f(i, j) = (r_1, \ldots, r_k) \implies \mathrm{ctr}(r_1) = P(i, j)$,
2. $f(1, j) = (r_1, \ldots, r_k) \implies \mathrm{ln}(r_s) = \vdash$ *for each* $s \in \{1, \ldots, k\}$,
   $f(i, 1) = (r_1, \ldots, r_k) \implies \mathrm{un}(r_s) = \top$ *for each* $s \in \{1, \ldots, k\}$,
   $f(m, j) = (r_1, \ldots, r_k) \implies \mathrm{rn}(r_s) = \dashv$ *for each* $s \in \{1, \ldots, k\}$,
   $f(i, n) = (r_1, \ldots, r_k) \implies \mathrm{dn}(r_s) = \bot$ *for each* $s \in \{1, \ldots, k\}$,
3. $f(i, j) C_\mathrm{h} f(i, j+1)$ *if* $j < n$,
   $f(i, j) C_\mathrm{v} f(i+1, j)$ *if* $i < m$

and there exists a linear ordering $\leq_g$ *of a set of rewritings* $S = \cup_{i=1}^{m} \cup_{j=1}^{n} S_{i,j}$*, where for each pair* (i, j) *s.t.* $f(i, j) = (r_1, \ldots, r_k)$ *we define*
$S_{i,j} = \{(i, j, r_1), (i, j, r_2), \ldots, (i, j, r_k)\}$*, and* $\leq_g$ *satisfies*

1. *for every witness* $\leq_c$ *of horizontal compatibility between* $f(i, j)$ *and* $f(i, j+1)$*,*

$$\begin{aligned}
(i, j, r) \leq_g (i, j+1, s) &\iff (r, \leftarrow) \leq_c (s, \rightarrow)\\
(i, j+1, r) \leq_g (i, j, s) &\iff (r, \rightarrow) \leq_c (s, \leftarrow)\\
(i, j, r) \leq_g (i, j, s) &\iff (r, \leftarrow) \leq_c (s, \leftarrow)\\
(i, j+1, r) \leq_g (i, j+1, s) &\iff (r, \rightarrow) \leq_c (s, \rightarrow)
\end{aligned}$$

2. *for the witness* $\leq_c$ *of vertical compatibility between* $f(i, j)$ *and* $f(i+1, j)$*,*

$$\begin{aligned}
(i, j, r) \leq_g (i+1, j, s) &\iff (r, \uparrow) \leq_c (s, \downarrow)\\
(i+1, j, r) \leq_g (i, j, s) &\iff (r, \downarrow) \leq_c (s, \uparrow)\\
(i, j, r) \leq_g (i, j, s) &\iff (r, \uparrow) \leq_c (s, \uparrow)\\
(i+1, j, r) \leq_g (i+1, j, s) &\iff (r, \downarrow) \leq_c (s, \downarrow)
\end{aligned}$$

Proof. Let $P \in \mathcal{L}(\mathcal{M})$ be a picture. Then there exists an accepting computation which can be described as a sequence of rewriting rules being applied at specific positions. By selecting rules applied to one fixed position (i, j), we obtain a reduction of $P(i, j)$ and the desired mapping $f(i, j)$. The ordering $\leq_g$ is obviously given by the order of the rewritings in the computation.

Conversely, let there be such mapping f and ordering $\leq_g$. We have to verify that performing rewritings from $\mathrm{dom}(\leq_g)$ in order suggested by $\leq_g$ constitutes a valid accepting computation.

Let $(i, j, r) \in \mathrm{dom}(\leq_g)$ and let the picture P' be P after performing, in the order given by $\leq_g$, exactly all rewritings $(i', j', r') \in \mathrm{dom}(\leq_g)$ such that $(i', j', r') <_g (i, j, r)$. Pick any neighbor of (i, j), for example the right one. If $j = n$, then $\widehat{P}(i, j+1) = \dashv$ and $\mathrm{rn}(r) = \dashv$ (because of mapping f requirement (2)). Otherwise, because $\leq_g$ respects orderings that witness compatibility with all neighboring reductions, $P'(i, j+1)$ must be equal to $\mathrm{rn}(r)$ (enforced by the definition of compatibility). The same argument applies to all of the neighbors as well as for the center of the rule. Therefore the rewriting and by induction the whole computation is valid. □

When we choose a matrix of reductions $M \in R(\mathcal{M})^{m,n}$ where the neighboring reductions are properly compatible, we can define an oriented graph $G = (V, \vec{E})$ where $V = \{(i, j, r) | r \in M_{ij}\}$ and $\vec{E}$ contains following types of edges:

1. $((i, j, r), (i, j, r'))$ if $M_{ij} = (r_1, \ldots, r, \ldots, r', \ldots, r_n)$
2. $((i, j, r), (i, j+1, r'))$ if $r \leq_c r'$, $((i, j+1, s), (i, j, s'))$ if $s \leq_c s'$ ($\leq_c$ being the respective horizontal compatibility witness)
3. $((i, j, r), (i+1, j, r'))$ if $r \leq_c r'$, $((i+1, j, s), (i, j, s'))$ if $s \leq_c s'$ ($\leq_c$ being the respective vertical compatibility witness)

We can observe that edges of a graph representing $\leq_g$ from the lemma (defined as $(V, \vec{E}')$ where $\vec{E}' = \{(v_1, v_2) \mid v_1 <_g v_2\}$) must be a superset of $\vec{E}$ and that any linear ordering whose graph is an extension of G meets all of the requirements of the lemma.

Such an ordering obviously cannot exist if there is an oriented cycle in G. Conversely, if there is no cycle, the graph has a topological ordering which can be used as the wanted $\leq_g$. This means that the existence of any ordering $\leq_g$ is equivalent to non-existence of an oriented cycle in G. This property can be verified by a simple depth-first search.

For a given restarting automaton, the size of $R(\mathcal{M})$ as well as any reduction by $\mathcal{M}$ is finite. Therefore the graph has a special form. Each position of the matrix corresponds to a limited number of vertices and edges in the graph only lead between vertices belonging to the same cell or between vertices of two neighboring cells. When suitably represented, a sgraffito automaton can perform depth-first search on such graphs (see [18]).

Now we can finally describe the sought sgraffito automaton $\mathcal{M}_S$. If the input picture is empty, it is immediately accepted. Otherwise, the automaton works in three stages.

1. In the first pass, a whole reduction is guessed at every position of the picture. This requires only one visit of each position (except for the edges of the picture).

2. In the second pass, neighbors are checked for local compatibility and the aforementioned graph G is built. To do this, every position only needs to be visited at most k times for some constant k.

3. Finally, a depth-first search of G is performed. If an oriented cycle is found, The picture is rejected. If the whole graph is explored without finding a cycle, the picture is accepted.

□

Lemma 5. *For all picture languages L in $\mathcal{L}(\mathsf{2SA})$, the language $L \cup \{\Lambda\}$ is in $\mathcal{L}(\mathsf{2LCRA})$.*

Proof. For any sgraffito automaton we can construct a limited context restarting automaton simulating it. This can be done virtually the same way as with restarting tiling automata. Therefore the following proof is an almost identical to the respective proof from [17].

Let $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_0, Q_F, \mu)$ be a $\mathsf{2SA}$. We describe a $\mathsf{2LCRA}$ $T = (\Sigma, \Gamma', \delta', \mu')$ such that $\mathcal{L}(\mathcal{T}) = \mathcal{L}(\mathcal{A})$. The idea is to simulate a computation of A over any input $P \in \Sigma^{*,*}$. If $\mathcal{A}$ scans the tape field f and the control unit is in state q, then $\mathcal{T}$ stores q into f. A set of rewriting rules is designed for changing the current configuration of $\mathcal{A}$ into a configuration after a single step of $\mathcal{A}$.

Let $k = |\Gamma|, m = \max\{k, 5\}$ and $I = \{0, \ldots, k\}$. Elements in Γ' are of six types:

1. $a \in (\Sigma \cup \mathcal{S}), \mu'(a) = mk + 4$, is an initial input symbol,

2. $(i,a) \in (I \times \Gamma)), \mu'((i,a)) = mi+3$, represents a field containing a, the head of $\mathcal{A}$ is not placed here, at most i instructions of $\mathcal{A}$ can be performed over the field,

3. $(i,a,q) \in (I \times \Gamma \times Q), \mu'((i,a,q)) = mi+1$, the same meaning of a and i as above, moreover, the head of $\mathcal{A}$ scans this field and the control unit is in the state q,

4. $(i,a,q,d) \in (I \times \Gamma \times Q \times \mathcal{H}), \mu'((i,a,q,d)) = mi$, the same meaning of a, i and q as above, moreover, $\mathcal{A}$ will move from this field in the direction d,

5. $(i,a,q,b) \in (I \times \Gamma \times Q \times \Gamma), \mu'((i,a,q,b)) = mi+2$, an auxiliary symbol with the meaning: $\mathcal{A}$ moved to this field containing a in the state q from a neighboring field by an instruction which writes b,

6. $\mathbf{0}, \mu'(\mathbf{0}) = 0$, a special zero weight symbol.

There is no loss of generality in assuming that $\mathcal{A}$ moves the head in each computation step. The specification of rewriting rules follows. For each $a \in \Sigma$, there is a rule creating the representation of the initial configuration:

$$\begin{array}{|c|c|}\hline \# & \top \\ \hline \vdash & a \\ \hline \end{array} \rightarrow \begin{array}{|c|c|}\hline \# & \top \\ \hline \vdash & (k,a,q_0) \\ \hline \end{array}.$$

For each instruction $(q,a) \rightarrow (q',a',R)$ in δ, rules matching the following patterns are added:

(i,a,q)	(j,b)	$\rightarrow$	(i,a,q,R)	(j,b)
(i,a,q)	b	$\rightarrow$	(i,a,q,R)	b
(i,a,q,R)	(j,b)	$\rightarrow$	(i,a,q,R)	(j,b,q',a')
(i,a,q,R)	b	$\rightarrow$	(i,a,q,R)	$(0,b,q',a')$
(i,a,q,R)	(j,b,q',a')	$\rightarrow$	$(i-1,a')$	(j,b,q',a')
$(i-1,a')$	(j,b,q',a')	$\rightarrow$	$(i-1,a')$	(j,b,q')

where $i,j \in I, 0 < i \leq k, b \in \Gamma \setminus \mathcal{S}, s,t \in \Sigma \cup \mathcal{S} \cup (I \times \Gamma)$. By writing an auxiliary symbol of the form (i,a,q,d) on the tape, we ensure that $\mathcal{T}$ cannot start to simulate simultaneously two instructions of $\mathcal{A}$ moving from a field in different (e.g. opposite) directions.

A special attention has to be paid to the situation, when the head of $\mathcal{A}$ moves outside P. We do not represent such a configuration, but rather the following configuration reached by applying a next instruction of the form $(q',\dashv) \rightarrow (q'',\dashv,L)$. Thus, the set of rules is completed by

$$\begin{array}{|c|c|}\hline (i,a,q) & \dashv \\ \hline \end{array} \rightarrow \begin{array}{|c|c|}\hline (i-1,a',q'') & \dashv \\ \hline \end{array}$$

The weight function μ has been defined to conform the rules. Similar rules are added also for the remaining instructions which move the head of $\mathcal{A}$ left, up or down.

The fact that $\mathcal{A}$ has reached an accepting state is manifested by rewriting any field to (i,a,q_f), where $a \in \Gamma, i \in I, q_f \in Q_F$. Such symbol is then rewritten to $\mathbf{0}$, which is subsequently propagated to the whole picture. $\square$

In addition to the equivalence between our model and the nondeterministic sgraffito automaton, we will now show that the correctness preserving variant of our model is equivalent to a deterministic sgraffito automaton.

Theorem 8. *For every picture language L, $L \in \mathcal{L}(\mathsf{2DSA}) \iff L \cup \{\Lambda\} \in \mathcal{L}(\mathsf{CP2LCRA})$.*

Proof. First, observe that the limited context restarting automaton simulating a given deterministic sgraffito automaton (constructed according to the description from the previous proof) is correctness preserving. The simulator is at all times bound to do one specific operation and cannot therefore deviate from an accepting computation.

To prove the other implication we will describe a deterministic sgraffito automaton simulating CP2LCRA $\mathcal{M} = (\Sigma, \Gamma, \delta, \mu)$. We will w.l.o.g. use a 2DSA that can move its head in eight directions. A diagonal movement of the head can be simulated by a four-way 2DSA using two movements (which raises the number of visits of some fields, but only by a constant).

The correctness preservation property ensures that whenever a rewriting is possible, it can be carried out without fear of straying from an accepting computation. The only problem for a sgraffito automaton is keeping track of places where the rewritings can be done. For that it needs a data structure that will hint the next position to check for possible rewriting. This structure can take advantage of the fact that once you make sure that a cell is not rewritable, it stays that way until at least one of its neighbors is rewritten.

To implement this structure, the automaton can keep an oriented tree that spans over all possibly rewritable positions. This tree has at most one vertex corresponding to each cell of the picture (meaning there is a chance of rewriting), and its edges can only lead between neighboring cells.

The automaton operates as follows:

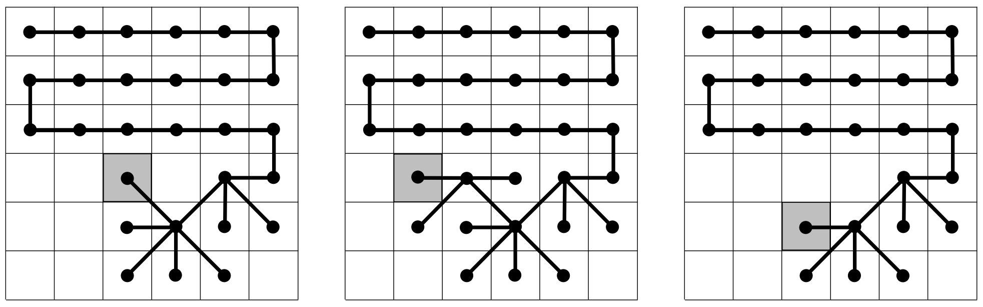
- First, the automaton marks all positions as potentially rewritable, i.e. constructs a tree that has vertices in all of them (a snakelike simple path), and moves its head to a leaf of that tree. This can be done in a single pass over the picture.

- After that, the rewriting of the picture is simulated. This is done in series of steps in which the automaton, if possible, rewrites the symbol at the current position and then moves to a next one.

  At the beginning of each step, the head of the automaton is positioned over a leaf of the tree. Here the automaton checks whether any rewriting can be done according to δ.

  If so, it is performed and all the neighbors that are not already part of the tree are connected to this leaf. Then the head moves to one of these new descendants (see Fig. 4.2b). If no children are spawned, the head stays.

  If not, the leaf is removed from the tree and the head moves to its parent. If this parent is not a leaf now, the head moves to one of the node's children (see Fig. 4.2c). These children are all leaves, because they have not been visited since their creation (after a child node is visited, it is not possible to return to its parent until the child is erased).



(a) Tree before a step. (b) Tree after a rewriting step. (c) Tree after a non-rewriting step.

Figure 4.2: An example of a step of a 2DSA simulating a CP2LCRA. At the beginning the automaton's head is above the leaf of the tree within the gray field (a). If a rewriting is carried out, the tree is extended (b). Otherwise the leaf is removed (c). Finally, the head moves to another leaf.

This process is repeated until there are no more leaves and the tree is erased completely.

- After the tree is reduced to zero vertices (meaning no more rewritings are possible), the automaton checks in a single pass whether all resulting symbols are of zero weight. If they are, the picture is accepted, otherwise it is rejected.

It remains to verify that in the second part of the computation the number of visits of each position is limited by a constant. Upon visiting a node of the tree, one of the three following actions is done:

- A leaf is removed (when no rewriting is possible),
- a leaf is transformed into an inner node (after rewriting) or
- an inner node loses one of its children (when returning from a removed leaf).

Each cell can be added to the tree (as a leaf) every time a rewriting is executed in the neighboring positions, which cannot happen more than $8|\delta|$ times. Obviously, the number of removals from the tree is the same.

A leaf can become an inner node of the tree only when a rewriting of its field is carried out (at most $|\delta|$ times). After its creation, the inner node is visited at most eight times (when returning from erased children) before becoming a leaf again.

The above calculations do not account for visits that are made when exploring a neighborhood of a cell (when matching a rule or looking for new cells to connect to a tree). Number of these visits for one position is however proportional to the number of actions performed in the neighboring positions and is thus still limited by a constant. $\square$

4.3.3 Two-dimensional restarting tiling automaton

It was shown in [17] that with an arbitrary scanning strategy, the restarting tiling automata are properly stronger than sgraffito automata (even in the class of one-dimensional inputs). This implies that 2RTA is more powerful than our model as well. However, a question remains whether there exists a "simple" scanning strategy ν such that $\mathcal{L}(\nu\text{-2RTA}) \subseteq \mathcal{L}(\text{2SA})$ where ν-2RTA denotes the class of restarting tiling automata using the scanning strategy ν.

One obvious argument against this inclusion is the difference in time complexity. In a picture of size (m, n), a sgraffito automaton can visit every field only constant times, so the whole computation has time complexity $O(mn)$. A restarting tiling automaton also performs $O(mn)$ rewritings, but every rewriting is preceded by scanning of $O(mn)$ tiles, so the total time complexity is $O(m^2n^2)$.

This disparity poses a problem for a potential simulation of a ν-2RTA $\mathcal{R}$ by some 2SA $\mathcal{S}$ (and a 2LCRA in extension). We unsuccessfully considered two approaches for such a simulation.

First approach is to simulate rewritings performed by $\mathcal{R}$ in the order in which $\mathcal{R}$ would perform them. To do that, $\mathcal{S}$ needs to implement a structure akin to a priority queue that hints possibly rewritable tile positions, in the order given by ν. At the beginning, this queue contains all $(m+1)(n+1)$ tile positions in the picture. Whenever a rewriting of $\mathcal{R}$ is simulated, up to four tiles that contain the rewritten field are added to the queue. So, in total, $O(mn)$ tile positions are enqueued during the computation.

$\mathcal{S}$ runs in $O(mn)$ time, so the described queue would have to perform both extraction of the first element and insertion of a new element in amortized $O(1)$ time. This seems unlikely to be achievable with the limited means of a sgraffito automaton.

Second approach we considered is to guess, or simulate the rewritings performed by $\mathcal{R}$ regardless of the correct order and then verify that they can be ordered to form a valid ν-2RTA computation. In a valid order, a rewriting is done if

- the rewritten tile matches the left-hand side of the associated rewriting rule and
- there is no rewritable tile position on the previous scanning path.

The problem lies with the second condition. The scanning strategy may cause that a rewriting of a tile (i, j) satisfying the first condition cannot be done by $\mathcal{R}$ because some rewritings of earlier scanned tiles lead to modification of the tile (i, j).

These relations between rewritings are not always easy to detect. As an example, consider a simulation where we guess a rewriting of the tile position $(i, j+1)$ by a rule

$$\begin{array}{|c|c|}\hline \mathbf{b} & \mathbf{c} \\ \hline \mathbf{b} & \mathbf{c} \\ \hline \end{array} \rightarrow \begin{array}{|c|c|}\hline \mathbf{b} & \mathbf{x} \\ \hline \mathbf{b} & \mathbf{c} \\ \hline \end{array}$$

and also a rewriting of the tile position (i, j) that precedes the position $(i, j+1)$ on the scanning path by a rule

$$\begin{array}{|c|c|}\hline \mathbf{a} & \mathbf{b} \\ \hline \mathbf{a} & \mathbf{b} \\ \hline \end{array} \rightarrow \begin{array}{|c|c|}\hline \mathbf{a} & \mathbf{x} \\ \hline \mathbf{a} & \mathbf{b} \\ \hline \end{array}.$$

There are two possibilities. The first one is that the latter rewriting can be performed independently of the former and is forced to be executed first by the scanning strategy. In this case the former rewriting cannot be executed, rendering our guess wrong.

The second possibility is that the first rewriting launches a series of changes that propagate throughout the picture, eventually leading to changing the tile (i, j) so as to enable the second rewriting. In this case our guess might have been correct. Tracking this kind of dependency requires non-local exploration of the picture, which could probably be done for one instance of this problem. However, these ambiguous situations can occur many times in one picture and performing the tracking for all of them might be impossible for a sgraffito automaton. If so, the two presented cases are generally indistinguishable and the simulation cannot function properly as a consequence.

In conclusion, the question of existence of a scanning strategy ν such that $\mathcal{L}(\nu\text{-}\mathsf{2RTA}) \subseteq \mathcal{L}(\mathsf{2SA})$ remains open.

Chapter 5

Recognition and learning algorithms

The main focus of this chapter is to present an algorithm for learning of automata. First, though, we need to address the problem of recognition of pictures by our model.

5.1 Recognition

By not using a scanning strategy, we have lost the ability to effectively simulate the automaton's operation by simple backtracking. Instead, we can take an alternative approach similar to the simulation via sgraffito automaton introduced in 4.3.2.

To implement this method, the recognition algorithm first generates a list of feasible reductions for every position of the picture and then repeatedly chooses sets of locally compatible reductions, until a set is found such that it constitutes a valid computation of 2LCRA. We will now describe each of these steps separately.

5.1.1 Enumerating feasible reductions

The input of the algorithm is comprised of a 2LCRA $\mathcal{M} = (\Sigma, \Gamma, \delta, \mu)$ and a picture P. In the first phase, we need to procure for every position (i, j) in P a list of possible reductions such that if there exists an accepting computation of $\mathcal{M}$ over P, then the list contains the reduction performed to rewrite the field (i, j) to zero. We also need to establish the relations of local compatibility between possible reductions of neighboring positions.

One way to obtain these lists is to generate the set of all reductions $R(\mathcal{M})$ and assign to each position (i, j) all reductions from $R(\mathcal{M})$ that are applicable to the symbol $P(i, j)$. However, with this method, the resulting lists of reductions are needlessly large, which might considerably slow down the next phase of the algorithm.

Instead, we try to generate for each position a smaller number of reductions that are relevant to that position in the context of P. We launch a sort of parallel simulation, where we, in a way, represent all possible states into which the picture can be rewritten by $\mathcal{M}$. This is done by trying to perform all possible rewritings in each position of P, which results in a set of all states to which the said position can be rewritten. Each state is linked to the rest of the picture only by keeping sets of compatible states of neighboring positions. These compatibility sets determine whether it is possible for a field in a given state to be rewritten further, and they are formed in the process of the parallel rewriting.

In accordance with the proof in 4.3.2, we call these states *partial reductions*. Unlike a *reduction*, a partial reduction does not have to lead to a zero-weight symbol. We represent a partial reduction p by a structure

$$p = \begin{array}{|c|c|c|}\hline N_1 & N_2 & N_3 \\ \hline N_4 & s & N_5 \\ \hline N_6 & N_7 & N_8 \\ \hline\end{array}$$

where $s \in \Gamma$ is the resulting symbol of the partial reduction and N_i are sets of locally compatible partial computations which can be performed within the respective neighboring positions.

At the beginning of the processing of a picture P, we define one partial reduction (consisting of zero rewritings) $c_{i,j}$ for each position $(i, j) \in \{0, \ldots, \text{rows}(P) + 1\} \times \{0, \ldots, \text{cols}(P) + 1\}$:

$$c_{i,j} = \begin{array}{|c|c|c|}\hline \{c_{i-1,j-1}\} & \{c_{i-1,j}\} & \{c_{i-1,j+1}\} \\ \hline \{c_{i,j-1}\} & \widehat{P}(i+1, j+1) & \{c_{i,j+1}\} \\ \hline \{c_{i+1,j-1}\} & \{c_{i+1,j}\} & \{c_{i+1,j+1}\} \\ \hline\end{array}\ .$$

In the cases of borders of the extended picture the sets of compatible partial reductions are empty for the non-existing neighbors. For example, at the left border of $\widehat{P}$ we have for $1 \le i \le \text{rows}(P)$ the partial computations

$$c_{i,1} = \begin{array}{|c|c|c|}\hline \emptyset & \{c_{i-1,0}\} & \{c_{i-1,1}\} \\ \hline \emptyset & \vdash & \{c_{i,1}\} \\ \hline \emptyset & \{c_{i+1,0}\} & \{c_{i+1,1}\} \\ \hline\end{array}\ .$$

From this initial setup, the algorithm begins to generate new partial reductions by extending the existing ones, using an operation that is analogous to performing a rewriting in P.

An *extension* of a partial computation

$$p = \begin{array}{|c|c|c|}\hline N_1 & N_2 & N_3 \\ \hline N_4 & s & N_5 \\ \hline N_6 & N_7 & N_8 \\ \hline\end{array}$$

by a rule

$$\begin{array}{|c|c|c|}\hline n_1 & n_2 & n_3 \\ \hline n_4 & s & n_5 \\ \hline n_6 & n_7 & n_8 \\ \hline\end{array} \rightarrow \begin{array}{|c|c|c|}\hline n_1 & n_2 & n_3 \\ \hline n_4 & s' & n_5 \\ \hline n_6 & n_7 & n_8 \\ \hline\end{array}$$

is a partial reduction

$$p' = \begin{array}{|c|c|c|}\hline N'_1 & N'_2 & N'_3 \\ \hline N'_4 & s' & N'_5 \\ \hline N'_6 & N'_7 & N'_8 \\ \hline\end{array}$$

where each $N'_i, i \in \{1, \ldots, 8\}$ is a set of all partial reductions from N_i that result in n_i. An extension can only be made if no N'_i is empty.

Upon creating an extension, we need to keep the symmetry of the compatibility relations. Therefore if a partial reduction

$$q = \begin{array}{|c|c|c|} \hline M_1 & M_2 & M_3 \\ \hline M_4 & n_i & M_5 \\ \hline M_6 & M_7 & M_8 \\ \hline \end{array}$$

becomes a compatible neighbor of p' and is added to N'_i, then p' must accordingly be added to M_{9-i} (for example, if q is a left neighbor of p', then q is added to N_4 and p', as a right neighbor of q, must be added to $M_{9-4} = M_5$).

Partial reductions generated for each position $(i, j) \in \{1, \ldots, \text{rows}(P)\} \times \{1, \ldots, \text{cols}(P)\}$ are kept in an oriented tree $G_{i,j} = (V, \vec{E})$ where V is the set of the partial reductions and $\vec{E}$ contains edges (p, q) such that q is an extension of p. The initial empty partial reductions are the roots of these trees. Note that every edge is associated with a rewriting rule that facilitated the respective extension and a path from a root of a tree to a node — a partial reduction p — gives us the sequence of rewriting rules of p.

To limit the size of these trees, we require that no partial reduction is extended by the same rule twice. If a partial reduction p gains a new compatible neighbor q which matches a rewriting rule that had already been used to extend p to p', then, instead of extending p again, q also becomes a compatible neighbor of p' (and vice versa). This operation is then recursively propagated to the relevant descendants of p'.

As a result of this, the size of one tree is independent of the size of the input picture and is at most exponential in the number of rules. Therefore, for a fixed automaton, the size of the tree is bounded by a constant, albeit possibly a big one.

The algorithm generates new partial reductions until there is no partial reduction among the existing ones that can be extended further by any rewriting rule. From the resulting trees we can extract the sought reductions (corresponding to leaves with a zero-weight result) and the local compatibility relations defined by the sets of compatible neighbors.

5.1.2 Choosing compatible reductions

In the second part of the algorithm we need to enumerate all possible combinations of reductions that might make up an accepting computation. This can be formulated as a constraint satisfaction problem (CSP; see e.g. [3]) where we have a variable for every position in P that selects a reduction for that position and the constraints are given by an enumeration of compatible pairs of neighboring reductions. A solution of this problem is a matrix of reductions M of size $(\text{rows}(P), \text{cols}(P))$, where M_{ij} is one of the listed possible reductions of $P(i, j)$ that is locally compatible with all of its neighbors in M.

For solving constraint satisfaction problems there exists a multitude of different techniques. The problem itself, however, is generally NP-complete.

5.1.3 Finding a valid computation

The final step of the algorithm is to verify that the found set of rewritings (given by the selected reductions in M) can be linearly ordered to form a valid computation. Every rewriting is represented here by the resulting partial reduction.

Each selected reduction M_{ij} represents a set of partial reductions (or rewritings) S_{ij} containing M_{ij} itself and all the ancestors of M_{ij} in the tree of partial reductions G_{ij} except for the root of G_{ij} (because the root represents no rewriting). On the set of all rewritings from all S_{ij} we build the graph containing the information about the local ordering of the neighboring rewritings.

If we take a partial reduction $p \in S_{ij}$ and restrict its list of compatible left neighbors N_l to the rewritings in $S_{i(j-1)}$, one of the three following cases occurs.

- $N_l \cap S_{i(j-1)}$ contains no partial reduction whose result matches the last rule used in p, meaning that the rule matches the original symbol $P(i, j)$. In this case we have $p < q$ for all $q \in N_l \cap S_{i(j-1)}$.
- The last rule used in p matches the result of the reduction $M_{i(j-1)}$ an thus we have $p > M_{i(j-1)}$.
- There is exactly one pair of rewritings $q_1, q_2 \in N_l \cap S_{i(j-1)}$ such that q_2 is an extension of q_1 and the result of q_1 matches the last rewriting rule used in p. In this case the local ordering must satisfy $q_1 < p < q_2$.

A similar principle applies to all eight neighboring positions. Apart from these inequalities, a partial reduction must also always come before its extension, so for all $p_1, p_2 \in S_{ij}$ such that p_2 is an extension of p_1 we have $p_1 < p_2$.

As with the simulation by a sgraffito automaton, we need verify that the graph incorporating the described orderings does not contain any oriented cycle. This can be done by a depth-first search. If no cycle is found, the rewritings can be topologically sorted, which gives us a possible order of rewritings in an accepting computation.

If a cycle is found, then the particular CSP solution is dropped and the algorithm moves to another one. When all solutions are depleted without finding any suitable one, the picture is rejected.

5.1.4 Using patterns

We have seen in Chapter 3 that it is usually easier to work with groups of rules represented by patterns rather than with single rules. For that reason it is advisable to use those patterns instead of rules when creating extensions of partial reductions in the first part of the algorithm.

Extending partial computation

$$p = \begin{array}{|c|c|c|}\hline N_1 & N_2 & N_3 \\ \hline N_4 & s & N_5 \\ \hline N_6 & N_7 & N_8 \\ \hline \end{array}$$

by a pattern

$$r = \begin{array}{|c|c|c|}\hline M_1 & M_2 & M_3 \\ \hline M_4 & s & M_5 \\ \hline M_6 & M_7 & M_8 \\ \hline \end{array} \to \begin{array}{|c|c|c|}\hline M_1 & M_2 & M_3 \\ \hline M_4 & s' & M_5 \\ \hline M_6 & M_7 & M_8 \\ \hline \end{array}$$

yields a partial reduction p' such that the respective sets of compatible neighbors N_i' contain those partial reductions that result in any of the symbols in M_i (and not just one particular symbol given by a rule).

In this algorithm we require that for each $i \in \{1,\dots,8\}$ the set $M_i \subseteq (\Gamma \cup \mathcal{S})$, apart from sentinels, only contains consecutive symbols w.r.t. the working alphabet Γ weighed by μ, meaning

$$(\forall s,t \in M_i)(\forall u \in \Gamma)\mu(s) \leq \mu(u) \leq \mu(t) \implies u \in M_i. \tag{5.1}$$

The usage of patterns affects the final part of the algorithm (the depth-first search) because it relaxes the local orderings of the rewritings. When using single rules, the rewritings of every pair of neighboring positions have a unique linear ordering and there is always only one way a given partial reduction can be inserted in the sequence of partial reductions of a neighbor in the ordering of the computation.

On the other hand, when we use patterns, a partial reduction can be inserted between several pairs of neighbor rewritings. Due to the condition (5.1) imposed on the patterns, the partial reductions $q_1, q_2, \dots, q_k \in S_{i(j-1)}$ ($k \in \mathbb{N}_0$) that result in symbols matching the last pattern used in a partial reduction $p \in S_{ij}$ form a consecutive string of rewritings such that q_{l+1} is an extension of q_l for all $l \in \{1,\dots,k-1\}$.

Because of this fact, we usually only need to set two inequalities for the ordering. First, if q_1 is an extension of a partial reduction q_0 such that the result of q_0 does not match the last pattern used in p, then we define $p > q_1$. Second, if there exists a partial reduction $q_{k+1} \in S_{i(j-1)}$ such that q_{k+1} is an extension of q_k, then we define $p < q_{k+1}$. The last special case is for $k = 0$, meaning that the p's last pattern only matches the input symbol $P(i, j-1)$. As with the similar case for single rules, we then have $p < q$ for all $q \in S_{i(j-1)}$. By imposing these inequalities for all neighbors, we ensure that in a computation based on the resulting global ordering, a pattern used for any rewriting matches all of the neighbors' contents at the time of the rewriting.

5.1.5 Time complexity

The time complexity of the presented algorithm depends significantly on the simulated automaton. On several examples from the previous chapter, the algorithm can be made to run in linear time, on others the recognition problem is NP–complete (for example the language of satisfiable CNF formulas). Here we will describe the worst case time complexity for a simulation of an automaton $\mathcal{M} = (\Sigma, \Gamma, \delta, \mu)$ on a picture P. We denote $n = (\text{cols}(P)\text{rows}(P)), m = \max_{a\in\Gamma}\mu(a)$ and by r the maximal number of reductions for any input symbol.

First, let us estimate the maximal number of reductions per position. In a reduction, each rewriting rule can be use at most once, and therefore there cannot be more than $2^{|\delta|}$ distinct reductions. On the other hand, Γ can contain up to $m+1$ symbols of distinct consecutive weights. Let us assume that for every non-zero symbol s there are k rules that rewrite s to the symbol t with $\mu(t) = \mu(s) - 1$. Then there are k^m reductions constructed from a set of km rules. This implies that for an arbitrary automaton there can be at least $\left(\frac{|\delta|}{m}\right)^m$ possible reductions.

With r reductions per symbol, a tree of partial reductions for one position has no more than rm nodes. When an extension of partial reduction is created, the new partial reduction can become a compatible neighbor of all rm neighboring partial reductions (which then have to be checked for possible extension by $|\delta|$)

rules). This gives a rough upper bound on the time complexity of the first part of the algorithm to be $O(|\delta|(rm)^2)$.

The second part, solving a CSP, is NP-complete, which (probably) makes the worst-case time complexity exponential in n.

The last part of the algorithm is a depth-first search of an oriented graph that has at most mn vertices (each of the n reductions consists of at most m rewritings) and of $O(mn)$ edges (because a rewriting is only compared to the rewritings that create and later destroy the 3-by-3 sub-picture matching the used rule). Therefore, for one search, the time complexity is $O(mn)$. This operation might have to be done for all solutions of the CSP.

5.2 Learning

In this section we will explore some possibilities for creating a learning algorithm for 2LCRA. The goal is to infer from sets of positive and negative samples $\langle S^+, S^- \rangle$, where $S^+, S^- \in \Sigma^{*,*}$, a 2LCRA $\mathcal{M} = (\Sigma, \Gamma, \delta, \mu)$ such that $S^+ \subseteq L(\mathcal{M})$ and $S^- \cap L(\mathcal{M}) = \emptyset$.

The inferred automaton should be as simple as possible so as not to be overfitted. Consider a simple automaton $\mathcal{M}_P$ that accepts a single picture P of size (m, n) by applying, from the top left corner to the bottom right corner, the rules

$$\begin{array}{|c|c|}\hline \gamma_{i-1,j-1} & \gamma_{i-1,j} \\ \hline \gamma_{i,j-1} & P(i,j) \\ \hline \end{array} \rightarrow \begin{array}{|c|c|}\hline \gamma_{i-1,j-1} & \gamma_{i-1,j} \\ \hline \gamma_{i,j-1} & \gamma_{i,j} \\ \hline \end{array}$$

for $i \in \{1, \ldots, m\}, j \in \{1, \ldots, n\}$, where $\gamma_{0,0} = \#$ and for all $k \in \{1, \ldots, m\}, l \in \{1, \ldots, n\}$, $\gamma_{0,l} = \top$, $\gamma_{k,0} = \vdash$ and $\gamma_{k,l}$ is an auxiliary symbol with zero weight. Union of such automata for each $P \in S^+$ (meaning an automaton that nondeterministically guesses P and simulates $\mathcal{M}_P$) accepts all positive samples and rejects all negative, but is obviously not a satisfactory result.

On the other hand, if we suitably throttle the possible power of the inferred automaton, it will have to base its computation on a general principle hinted by the examples. This limitation can be imposed by fixing the size of the working alphabet to a small value $k \geq |\Sigma|$.

Even for a small k, the number of possible automata is very big. As an example, we can count the number of possible automata that

- have a fixed injective weight function and
- only accept pictures with more than one row and more than one column.

The only structure that is still variable is the set of rewriting rules. To count the number of distinct rules we first consider all possible configurations of neighbors of a field.

1. A field in one of the four corners has three arbitrary neighbors, so there are $4k^3$ corner configurations,
2. a field on one of the four borders has five neighbors, so we have $4k^5$ border configurations and
3. all other fields have eight neighbors, resulting in another n^8 configurations.

A valid rewriting rule can be assembled from any configuration of neighbors and any pair of symbols from Γ (where the symbol with higher weight is rewritten to the other one). This makes the number of possible rewriting rules

$$d = \binom{k}{2}(4k^3 + 4k^5 + k^8)$$

From all these rules we can build 2^d different sets to define 2^d different automata. For $k = 2$ (for example an automaton with two input symbols and no auxiliary symbols) this number is 2^{416}, for $k = 3$ (for example two input symbols and one auxiliary symbol) it is 2^{22923}. Obviously, we need to make some further simplifications in order to have a smaller search space.

First simplification lies in using rewriting rules in the cross format, which reduces the number of neighbors to four. We also do not seek single rules, but rule patterns

$$\begin{array}{|c|c|c|} \cline{2-2} \multicolumn{1}{c|}{} & N_1 & \multicolumn{1}{c}{} \\ \hline N_2 & N_c & N_3 \\ \hline \multicolumn{1}{c|}{} & N_4 & \multicolumn{1}{c}{} \\ \cline{2-2} \end{array} \rightarrow \begin{array}{|c|c|c|} \cline{2-2} \multicolumn{1}{c|}{} & N_1 & \multicolumn{1}{c}{} \\ \hline N_2 & s & N_3 \\ \hline \multicolumn{1}{c|}{} & N_4 & \multicolumn{1}{c}{} \\ \cline{2-2} \end{array}$$

with lists of *admissible symbols* for each position (as outlined in Chapter 3).

To find automata defined by these patterns, we encode them in bit vectors of fixed length and perform a randomized search among these vector representations.

5.2.1 Automaton representation

For easier handling, we represent an automaton $\mathcal{M}_B = (\Sigma, \Gamma, \delta, \mu)$ by a bit vector $B = (b_1, b_2, \ldots, b_d)$ (where $(\forall i \in \{1, \ldots, d\}) b_i \in \{0, 1\}$) of some fixed length d (which depends on how a conversion from vectors to automata is implemented).

For our search algorithm, we impose two conditions for the bit-vector representations.

(Bit 1) If $B = (1, 1, \ldots, 1)$, then $L(\mathcal{M}_B) = \Sigma^{*,*}$ (which implies $S^+ \subseteq L(\mathcal{M}_B)$).

(Bit 2) For any $B_1 = (b_1, b_2, \ldots, b_d)$ and $B_2 = (c_1, c_2, \ldots, c_d)$,
if $(\forall i \in \{1, \ldots, d\}) b_i \leq c_i$, then $L(\mathcal{M}_{B_1}) \subseteq L(\mathcal{M}_{B_2})$.

There are many ways in which we can perform the conversion between automata and bit vectors. We want all the bit vectors to be of the same length, so we use a fixed architecture of automata where

- Σ contains all symbols appearing in pictures from S^+ and S^-,
- Γ contains Σ and a fixed number of auxiliary symbols,
- δ is described by a fixed number (which is tied to the size of Γ) of rule patterns and
- μ is fixed to be 1 for input symbols and 0 for auxiliary symbols.

Our definition of μ results in automata that can only perform one rewriting per field.

We propose three different architectures of encoded automata. In all of them, the complexity of the automata is regulated by a single input parameter $k \in \mathbb{N}$. Given k, we build implicit Σ, Γ and μ and define the number of rule patterns k_p. Each pattern p_i is represented by d_i bits $(b^i_1, \ldots, b^i_{d_i})$. An automaton $\mathcal{M}_B$ is then represented by a concatenation of these patterns' vectors:

$$B = (b^1_1, \ldots, b^1_{d_1}, b^2_1, \ldots, b^2_{d_2}, \ldots, b^{k_p}_1, \ldots, b^{k_p}_{d_{k_p}})$$

The proposed architectures are outlined below.

(Arch 1) For a given k, Γ contains k zero-weight auxiliary symbols $\{s_1, \ldots, s_k\}$ and each s_i is associated with a single pattern

$$p_i = \begin{array}{ccc} \cline{2-2} & \multicolumn{1}{|c|}{N_1} & \\ \hline \multicolumn{1}{|c|}{N_2} & \multicolumn{1}{c|}{N_c} & \multicolumn{1}{c|}{N_3} \\ \hline & \multicolumn{1}{|c|}{N_4} & \\ \cline{2-2} \end{array} \rightarrow \begin{array}{ccc} \cline{2-2} & \multicolumn{1}{|c|}{N_1} & \\ \hline \multicolumn{1}{|c|}{N_2} & \multicolumn{1}{c|}{s_i} & \multicolumn{1}{c|}{N_3} \\ \hline & \multicolumn{1}{|c|}{N_4} & \\ \cline{2-2} \end{array}$$

where $N_1, N_2, N_3, N_4 \in (\Gamma \cup \mathcal{S})$, $N_c \in \Sigma$. For every pattern, we encode each $N_i, i \in \{1, 2, 3, 4\}$ in $|\Gamma| + 1$ bits $(b^i_0, \ldots, b^i_{|\Gamma|})$ so that

- $b^i_0 = 1 \implies \mathcal{S} \subseteq N_i, b^i_0 = 0 \implies \mathcal{S} \cap N_i = \emptyset$,
- $(\forall j \in \{1, \ldots, |\Gamma|\}) \quad b^i_j = 1 \iff \gamma_j \in N_i$, where $\{\gamma_1, \ldots, \gamma_{|\Gamma|}\} = \Gamma$.

Note that we use a single bit to represent a possibility of any sentinel — the kind of the sentinel is always decided by its relative position in the pattern.

Similarly, we encode N_c in $|\Sigma|$ bits $(b_1, \ldots, b_{|\Sigma|})$ so that

$$(\forall j \in \{1, \ldots, |\Sigma|\}) b_j = 1 \quad \iff \quad \sigma_j \in N_c, \text{ where } \{\sigma_1, \ldots, \sigma_{|\Sigma|}\} = \Sigma.$$

Every pattern is then represented by a concatenation of these five vectors. One pattern is encoded in $4(|\Gamma| + 1) + |\Sigma|$ bits, which makes the length of the representation of the automaton

$$\begin{aligned} d &= k\,[4\,(|\Gamma| + 1) + |\Sigma|] \\ &= k\,[4(|\Sigma| + k + 1) + |\Sigma|] \\ &= 4k^2 + (5|\Sigma| + 4)k. \end{aligned}$$

(Arch 2) To obtain even simpler automata, we can reduce the auxiliary alphabet to a single symbol $\mathbf{z}$ and have k patterns

$$\begin{array}{ccc} \cline{2-2} & \multicolumn{1}{|c|}{N_1} & \\ \hline \multicolumn{1}{|c|}{N_2} & \multicolumn{1}{c|}{N_c} & \multicolumn{1}{c|}{N_3} \\ \hline & \multicolumn{1}{|c|}{N_4} & \\ \cline{2-2} \end{array} \rightarrow \begin{array}{ccc} \cline{2-2} & \multicolumn{1}{|c|}{N_1} & \\ \hline \multicolumn{1}{|c|}{N_2} & \multicolumn{1}{c|}{\mathbf{z}} & \multicolumn{1}{c|}{N_3} \\ \hline & \multicolumn{1}{|c|}{N_4} & \\ \cline{2-2} \end{array}$$

which are encoded similarly as before. By this simplification, we can reduce the length of an automaton representation to

$$\begin{aligned} d &= k\,[4\,(|\Gamma| + 1) + |\Sigma|] \\ &= k\,[4(|\Sigma| + 1 + 1) + |\Sigma|] \\ &= (5|\Sigma| + 8)k. \end{aligned}$$

(Arch 3) To use a different approach for the encoding, we can for the given k define $k|\Sigma|$ auxiliary symbols $s_{\sigma i}$ where $\sigma \in \Sigma, i \in \{1, \dots, k\}$ that are identified by an input symbol and one of k categories. Again, there is a rule pattern for each $s_{\sigma i}$, but now the center of the rule is specified:

$$p_{\sigma i} = \begin{array}{|c|c|c|} \cline{2-2} \multicolumn{1}{c|}{} & N_1 & \multicolumn{1}{c}{} \\ \hline N_2 & \sigma & N_3 \\ \hline \multicolumn{1}{c|}{} & N_4 & \multicolumn{1}{c}{} \\ \cline{2-2} \end{array} \rightarrow \begin{array}{|c|c|c|} \cline{2-2} \multicolumn{1}{c|}{} & N_1 & \multicolumn{1}{c}{} \\ \hline N_2 & s_{\sigma i} & N_3 \\ \hline \multicolumn{1}{c|}{} & N_4 & \multicolumn{1}{c}{} \\ \cline{2-2} \end{array}$$

Every set of admissible neighbors N_i is encoded by $|\Sigma| + k + 2$ bits $(b_0, b_1, \dots, b_{|\Sigma|}, c_0, c_1, \dots, c_k)$ which define N_i in the following way.

- For sentinels we have $b_0 = 1 \implies \mathcal{S} \subseteq N_i, b_0 = 0 \implies \mathcal{S} \cap N_i = \emptyset$.
- For $\sigma_l \in \{\sigma_1, \dots, \sigma_{|\Sigma|}\} = \Sigma$ we define $\sigma_l \in N_i \iff b_l = 1 \wedge c_0 = 1$.
- For $s_{\sigma j} \in \Gamma \setminus \Sigma$, where $\sigma = \sigma_l \in \{\sigma_1, \dots, \sigma_{|\Sigma|}\} = \Sigma$, we define $s_{\sigma j} \in N_i \iff b_l = 1 \wedge c_j = 1$.

Informally, this means that a set of admissible neighbors is described by a list of acceptable input symbols (or symbols rewritten from them) and a list of acceptable categories. This approach performs well in practice, even though it generates larger bit representations of automata.

We have $k|\Sigma|$ patterns, each represented by $4(|\Sigma|+k+2)$ bits, so a complete automaton is represented by

$$\begin{aligned} d &= 4k|\Sigma|(|\Sigma| + k + 2) \\ &= 4k^2 + 4|\Sigma|^2k + 8k \end{aligned}$$

bits.

Of course, many other possible architectures could be devised. Here we have only listed some of the most straightforward concepts.

5.2.2 Automaton evaluation

When performing a local search we must evaluate automata in order to choose a best specimen from a set of candidates.

The main measure of the quality of an automaton $\mathcal{M}$ is how well it accepts pictures from S^+ and rejects pictures from S^-, which is expressed by a function

$$f_s(\mathcal{M}) = |S^+ \cap L(\mathcal{M})| - |S^- \cap L(\mathcal{M})|.$$

There are other aspects that we can measure:

- The simplicity of the automaton from the human point of view. For a human, the rules are more comprehensible if the sets of admissible symbols in the patterns contain complete sets of symbols (meaning the symbol in the respective position is not important) or either very few symbols or almost all of them (which gives the position a clearer meaning).

  We rate every set of admissible symbols N that can only contain symbols from some $N_a \subseteq \Gamma \cup \mathcal{S}$ by

$$f_n(N) = \begin{cases} c_a & \text{if } N = N_a \\ c_b \left(\frac{|N|}{|N_a|}\right)^2 & \text{if } \frac{|N_a|}{2} \le |N| \le |N_a| \\ c_b \left(\frac{|N_a|-|N|}{|N_a|}\right)^2 & \text{if } |N| < \frac{|N_a|}{2} \end{cases}$$

where c_a, c_b are constants subject to tuning. In our experiments, we used the values $c_a = c_b = 1$.

The total rating $f_n(\mathcal{M})$ is then a sum of $f_n(N)$ of all sets N of admissible symbols in all positions of all rule patterns.

- The simplicity of the automaton from the computer's point of view. We want the resulting automata to be as easy to simulate as possible, so we factor in the time complexity of the hardest part of the simulation — solving the underlying CSP problem. For a picture P, we can estimate this complexity (denoted by $f_c(P)$) in two ways.

  One possibility is to actually count the steps performed by the employed CSP solving algorithm. The downside of this approach is that the estimate very much depends on the used CSP solver.

  The second possibility is to estimate an upper bound of the complexity based on the counts of reductions that are produced by the recognition algorithm in every position of P. Denoting by r_{ij} the number of reductions in a position (i, j), there are

  $$f_c(P) = \prod_{(i,j)\in I} r_{ij}$$

  (where $I = \{1, \dots, \text{rows}(P)\} \times \{1, \dots, \text{cols}(P)\}$) combinations of reductions that the CSP solver can potentially explore. Obviously, this estimate is usually somewhat exaggerated.

  For an automaton $\mathcal{M}$, we rate its simplicity by

  $$f_c(\mathcal{M}) = -\sum_{P \in S^+ \cup S^-} f_c(P).$$

To compare automata according to these measures, we combine them in a tuple

$$f(\mathcal{M}) = (f_s(\mathcal{M}), f_n(\mathcal{M}), f_c(\mathcal{M})),$$

which we sort by a lexicographical ordering.

5.2.3 Search

Now we can describe the search procedure itself.

We confine the search space to bit vectors B such that $S^+ \subseteq L(\mathcal{M}_B)$ (i.e. the represented automata accept all positive samples). At the beginning we take the vector $(1, 1, \dots, 1)$, which must represent an automaton accepting $\Sigma^{*,*}$ because of the condition (Bit 1). Then we perform a series of steps in which we change some elements of the vector to zero, thus reducing the power of the represented automaton.

Evaluating an automaton is a relatively complicated operation (as it includes simulation over all samples), so we aim to change more than one bit in one step using as little effort as possible.

To do that, we represent a bit vector of length d by a pair (p, z) where p is a permutation of d elements and $0 \leq z \leq d$ is an integer. This pair is interpreted as a vector $B(p, z) = (b_1, \ldots, b_d)$ where $b_{p(i)} = 0$ for $i \leq z$ and $b_{p(i)} = 0$ for $i > z$.

Due to the condition (Bit 2) for the bit vectors, we have for a permutation p the following inclusions.

$$L(\mathcal{M}_{B(p,d)}) \subseteq L(\mathcal{M}_{B(p,d-1)}) \subseteq L(\mathcal{M}_{B(p,d-2)}) \subseteq \cdots \subseteq L(\mathcal{M}_{B(p,0)})$$

Because of that, we can define z_p as an index of the least powerful automaton that still accepts all pictures in S^+, i.e. $L(\mathcal{M}_{B(p,i)}) \supseteq S^+$ for all $i \leq z_p$ and $L(\mathcal{M}_{B(p,i)}) \not\supseteq S^+$ for all $i > z_p$. For a given permutation p, we can find z_p by the interval bisection method using at most $\log_2 d$ steps consisting of simulating an automaton over all positive picture samples.

In every step of the search algorithm, we have a current bit vector $B = B(p, z_p)$ (except for the beginning, when we have $B(p, 0)$). We try to modify B by several ways by changing p to some permutations $p_1, p_2, \ldots, p_b$ (where b is a tunable constant) using random permutations $q_1, q_2, \ldots, q_b$ of $(d - z_p)$ elements:

$$p_j(i) = \begin{cases} p(i), & i \leq z_p \\ p(z_p + q(i)), & i > z_p \end{cases}$$

In other words, we scramble the end of p that corresponds to ones in $B(p, z_p)$. For all the new permutations p_j we find z_{p_j} and evaluate the corresponding automata $\mathcal{M}_B(p_j, z_{p_j})$. If the best of these automata is an improvement over the current $\mathcal{M}_B$, we take it as the new current automaton.

If the algorithm fails to improve the current automaton $\mathcal{M}_B$ in b_f consecutive steps (b_f being a tunable constant), we assume that a locally best result has been reached. If this result satisfies $L(\mathcal{M}_B) \cap S^- = \emptyset$, we return it as the solution. Otherwise the search is restarted with $B = (1, \ldots, 1)$.

5.2.4 Output Simplification

Due to the random nature of the search, the output rule sets usually seem needlessly complex and confusing for a human. To make them more readable, we can try to simplify it by two kinds of changes:

- Replacing a set of admissible symbols in a pattern by a set of all symbols — this symbolizes that a content of the respective field is in fact not important to the rewriting (which occurs commonly in man-made automata).

- Removing symbols from a set of admissible symbols — if some symbols are unnecessary, then it is obviously better not to include it and obtain as small set as possible.

These changes can be made in any position in any pattern, but only if they do not adversely affect the performance of the automaton. When we perform a simplification of an automaton $\mathcal{M}$, we must verify that the simplified automaton $\mathcal{M}'$ satisfies

$$L(\mathcal{M}') \supseteq (L(\mathcal{M}) \cap S^+) \land (L(\mathcal{M}') \cap S^-) \subseteq L(\mathcal{M}).$$

If the condition is not met, the particular simplification cannot be made.

5.2.5 Speedup

In the process of searching, the sample pictures need to be checked for acceptance by many generated automata. These automata are very random, which means that there is no guarantee that their simulation (especially the CSP-solving part) will not take a long time.

We can make an assumption that the sought automaton is relatively simple to simulate, i.e. the time complexity of the associated CSP is polynomial in the size of the input picture. With this assumption we can set a polynomial time limit for every computation of the employed CSP solver and if the limit is exceeded, the computation concludes with a "not accepted" result for positive samples and with a "not rejected" result for negative samples.

We can use a different approach by considering a deterministic variant of our model.

Enforcing determinism requires re-addition of some sort of scanning strategy. Here we take inspiration from 2OTA (see 1.2) to implement scanning of a picture in diagonal lines $\{P(i,j) \mid i+j = t-1\}$ for steps t from 1 to $\text{rows}(P)+\text{cols}(P)-1$. A cycle of this model consists of finding the first diagonal line where a rewriting is possible, performing all possible rewritings on that diagonal line in parallel (this is possible when using rules in the cross format) and restarting.

In our case we also need an ordering of the rule patterns so that when more patterns are matched at one position, one is deterministically selected.

This ordering leads to breaking the condition (Bit 2) for the bit representations of all our architectures. For example, an automaton $\mathcal{M}$ can have the first rule pattern such that it rewrites parts of a picture P to a symbol that is incompatible with other rules and blocks the computation. In that case, P is rejected. However, by changing some elements of the bit representation of $\mathcal{M}$ to zero we can disable that pattern altogether and enable rewriting of P to zero by the other patterns.

Despite this shortcoming, our experiments show that the search algorithm still performs well using this deterministic variant. In this case the bisection method, used in the search to find the least powerful automaton from a sequence of automata, loses its consistency and effectively becomes a heuristic that finds some weakened automaton, though possibly not the weakest one there is.

Another way we can reduce the complexity of the computation is to use correctness preservation. This property cannot be verified for an arbitrary automaton, but we can assume that the sought automaton is correctness preserving. With this assumption we can, for all generated automata, run computations over positive samples using the fast recognition algorithm for CP2LCRA (see 2.2). If a picture is not accepted using this algorithm, we assume that it is not accepted at all. The usage of this algorithm causes, like in the case of determinism, breaking of the condition (Bit 2) because here we also try to use the rewriting rules in some fixed order.

The correctness preservation of any automaton is, however, not guaranteed, so the negative samples have to be processed normally to ensure that there exist no accepting computations. This disparity between processing of the positive and of the negative samples is somewhat balanced by the fact that our search procedure performs more computations over positive samples than over negative samples.

5.2.6 Extension

We have focused on a class of very simple automata that perform at most one rewriting in each tape field. In theory, a similar algorithm could be used to infer arbitrarily powerful automata by using an architecture with different arrangement of the weight function. For example, for a fixed number of auxiliary symbols k, one could define the weight function to be $\mu(s) = k$ for all $s \in \Sigma$ and $\mu(s_i) = i - 1$ for $\{s_1, s_2, \ldots, s_k\} = \Gamma \setminus \Sigma$. Then we could define rewriting rules using one rule pattern per auxiliary symbol s_i

$$\begin{array}{ccc} & \boxed{N_1} & \\ \boxed{N_2} & \boxed{N_c} & \boxed{N_3} \\ & \boxed{N_4} & \end{array} \rightarrow \begin{array}{ccc} & \boxed{N_1} & \\ \boxed{N_2} & \boxed{s_i} & \boxed{N_3} \\ & \boxed{N_4} & \end{array}$$

where $N_1, N_2, N_3, N_4 \in \Gamma \cup \mathcal{S}$ and $N_c \in \Sigma \cup \{s_1, s_2, \ldots, s_{i-1}\}$.

With this setup, we can represent any 2LCRA $\mathcal{M} = (\Sigma', \Gamma', \delta', \mu')$ using the following conversions.

- Different weighs of symbols in Σ' can be simulated by rewriting input symbols to some new auxiliary symbols that represent them while having the desired weights.

- If there are several zero-weight symbols, we can add a rule that rewrites them to a single zero-weight symbol. In order to do this, the original weights of symbols in Γ' have to be incremented by one.

- If more patterns are necessary to describe all the rules resulting in a symbol $s \in \Gamma'$, we can replace s by a sufficiently large set of new symbols (which are all treated as s in the other rules) and have each pattern result in a different symbol from this set.

- Finally, we observe that any automaton $(\Sigma, \Gamma, \delta, \mu)$ with a single zero-weight symbol can be modified to use a weight function μ'' such that

  $$(\forall s_1, s_2 \in \Gamma \setminus \Sigma)\mu''(s_1) \neq \mu''(s_2).$$

  This μ'' is induced by any topological ordering of a directed graph (Γ, E) where $E = \{(s_1, s_2) \mid \exists N \in (\Gamma \cup \mathcal{S})^8, (s_1, s_2, N) \in \delta\}$. This graph is acyclic, so the ordering always exists.

Using a conversion to a bit-vector similar to (Arch 1), we obtain a representation that is lengthened by the bits that represent possibilities of auxiliary symbols in the centers of the rules. Its length for a parameter k is

$$\begin{aligned} d &= 4k^2 + (5|\Sigma| + 4)k + \sum_{i=0}^{k-1} i \\ &= 4k^2 + (5|\Sigma| + 4)k + \frac{k}{2}(k-1) \\ &= \frac{9}{2}k^2 + (5|\Sigma| + \frac{7}{2})k. \end{aligned}$$

Apart from increasing the length of the bit representation, this architecture also rapidly increases the number of possible reductions for one field. In the worst case there can exist for every set of non-zero auxiliary symbols a reduction that uses exactly the symbols from this set (and the zero), which gives us a total of 2^{k-1} distinct reductions.

Another way to increase the power of the algorithm is to consider intersections of several inferred automata By an intersection of automata $\mathcal{M}_1, \ldots \mathcal{M}_n$ we mean an automaton $\mathcal{M}_I$ such that $\mathcal{M}_I$ accepts a picture P if every $\mathcal{M}_j, j \in \{1, \ldots, n\}$ accepts P. We represent $\mathcal{M}_I$ simply by the list of the intersected automata.

The proposed search algorithm only explores automata accepting all positive samples, so by an intersection we get an automaton that also accepts all the positive samples yet might reject more of the negative samples.

Using unlimited number of intersecting automata may lead to over-fitting so we set a maximum number of them, denoted by m. With that we can implement the following simple accumulator of automata.

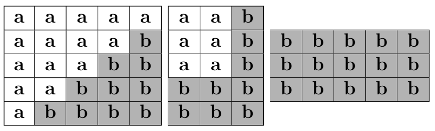
At the beginning, the accumulator is empty. Every time the search algorithm is about to restart, the inferred automaton (that rejects some of the negative samples) is committed to the accumulator's pool of automata. If the pool now contains $m + 1$ items, one of them is removed so that the intersection of the rest rejects the most negative samples. When every negative sample is rejected by at least one automaton in the current pool, we declare the result to be an intersection of all the pooled automata.

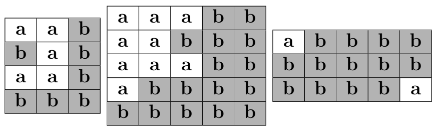
Chapter 6

Testing

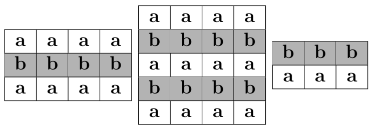
In this chapter we present the results achieved with our implementation of the learning algorithm. We tested both the speed of the algorithm and the quality of the inferred automata. For the testing we used six basic picture languages we describe below. All of them use a two-letter alphabet $\{\mathbf{a}, \mathbf{b}\}$.

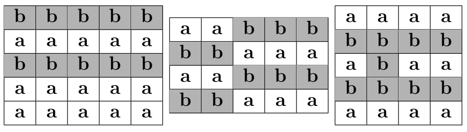
L_1 is the language of pictures where the letters are "sorted" in the sense that whenever a position (i, j) in $P \in L_1$ contains **a**, all positions (i', j'), $1 \leq i' \leq i, 1 \leq j' \leq j$ also contain **a**.


(a) Positive examples of L_1.

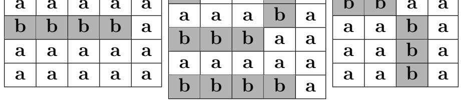

(b) Negative examples of L_1.

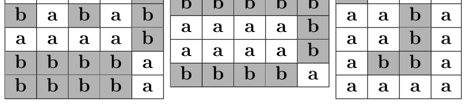
L_2 is the language of pictures with alternating horizontal stripes of both symbols, i.e. for a picture P and all positions (i, j), $2 \leq i \leq \text{rows}(P), 2 \leq j \leq \text{cols}(P)$, we have $P(i, j) = P(i, j-1)$ and $P(i, j) \neq P(i-1, j)$ (and, of course, $P(1, 1) = P(1, 2) \neq P(2, 1)$).


(a) Positive examples of L_2.

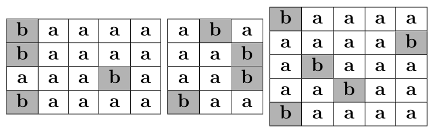

(b) Negative examples of L_2.

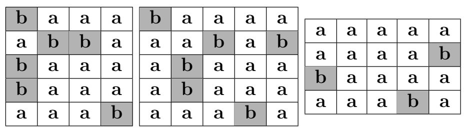
L_3 contains pictures where **b**'s form horizontal or vertical lines (of unit width) that do not touch each other.


(a) Positive examples of L_3.

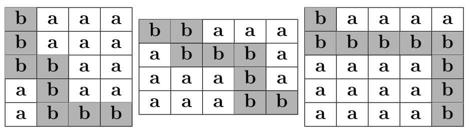

(b) Negative examples of L_3.

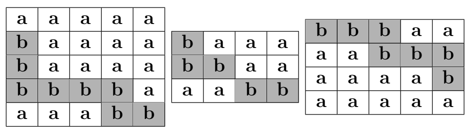
L_4 contains pictures with exactly one **b** in each row.


(a) Positive examples of L_4.

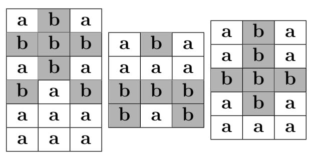

(b) Negative examples of L_4.

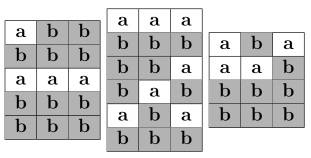
L_5 is the language of pictures with a path of **b**'s leading from the upper left corner to the bottom right corner. These paths only turn right or down.


(a) Positive examples of L_5.


(b) Negative examples of L_5.

L_6 contains pictures P such that $\text{cols}(P) = 3$ and the first column of P is equal to the last column (i.e. $P(i,1) = P(i,3)$ for all i, $1 \le i \le \text{rows}(P)$).


(a) Positive examples of L_6.


(b) Negative examples of L_6.

In the following table we list the complexity parameters which we use for these languages when utilizing the automaton architectures described in 5.2.1. k denotes the input parameter for each architecture and d denotes the length of the respective bit vector representation.

	Arch 1		Arch 2		Arch 3	
	k	d	k	d	k	d
L_1	1	18	1	18	1	40
L_2	2	44	2	36	1	40
L_3	3	78	3	54	2	96
L_4	3	78	3	54	2	96
L_5	3	78	3	54	3	168
L_6	4	120	4	72	4	256

6.1 Setting the parameters

In this section, we focus on tuning of the variable aspects of the learning algorithm in order to maximize its speed. For now, the only measure of the quality of the algorithm is the average time of finding an arbitrary solution.

The parameters we try to optimize are

- the variant of the 2LCRA model (discussed in 5.2.5),

- the automaton architecture (discussed in 5.2.1) and
- the parameters of the search algorithm (discussed in 5.2.3) — the number of generated modifications of the current automaton in each step (denoted by b) and the number of unsuccessful consecutive steps allowed before forcing a restart (denoted by b_f).

We tune each of these aspects separately with a possibly incorrect assumption that by changing one parameter, we do not dramatically change the impact of the others.

The learning algorithm works in cycles. Each cycle terminates with a restart, after which the search begins anew without retaining any information from the previous cycles. Thus the search can be viewed as a sequence of independent experiments, each one of which has some (small) constant probability of success, meaning that the time of finding a result is a random variable with geometric distribution.

This fact has two implications. The first is that if we want to estimate the speed of the algorithm accurately, we need to average the times of many runs because of the great variance of this distribution.

The other implication is that it is beneficial to run the algorithm in several independent threads at once. By running m threads, we increase the chance of success at any time m-fold, so the average time needed to find a solution is m times shorter as a result.

To compare different settings of the algorithm, we randomly generated a training set of 50 positive and 50 negative samples for every test language. Then we ran, one by one, all of the tested configurations of the algorithm on this set in eight parallel threads for a fixed time t. We measured the performance by counting the cycles in which a solution has been found.

In the first experiment, we compared the variations of the model of the automata used for recognition of the samples. We tested three approaches:

(M1) All samples are processed by an unmodified 2LCRA with unlimited computation time.

(M2) Positive samples are processed by a 2LCRA with the assumption that the automaton is correctness preserving, and negative samples are processed by the full 2LCRA recognition algorithm with its running time limited by a polynomial $p(mn)$ of the size of the input picture (m, n). In this experiment, we set the limit as $p(mn) = mn\sqrt{mn}$.

(M3) The deterministic variant of 2LCRA is used for recognition of both positive and negative samples.

For this comparison, we used the architecture (Arch 3) and fixed the search parameters to $b = b_f = 4$. The results are shown in the following table. For a given language and a given model, the displayed value is the number of solutions found in the time interval indicated by t. Obviously, a higher number means a better performance.

L	t	M1	M2	M3
L_1	10 s	15	50	423
L_2	10 s	18	49	411
L_3	100 s	3	18	80
L_4	100 s	0	1	9
L_5	100 s	1	2	20
L_6	200 s	1	0	9

Even though the values show some variance in repeated runs, the dominance of the deterministic model appears to be indisputable. Apparently, the determinisation of our model is instrumental in fast learning.

The second experiment compares the three architectures described in 5.2.1. For this comparison we use the deterministic variant of our model and, as before, we fix the search parameters to $b = b_f = 4$.

The following table shows the results in the same form as the previous one, i.e. each value is a number of found solutions in time t for the given language, using the respective architecture.

L	t	Arch 1	Arch 2	Arch 3
L_1	10 s	50	61	423
L_2	10 s	52	23	411
L_3	100 s	3	0	80
L_4	100 s	2	1	9
L_5	100 s	10	9	20
L_6	200 s	1	3	9

We can see that the third architecture is clearly the most useful, at least for the tested languages. This is despite the fact that this architecture generates longer bit representations of automata, thus exponentially expanding the search space. Conversely, the second architecture using the shortest bit representations has the worst performance. This may be because it is usually harder to design for a given language an automaton that uses only one auxiliary symbol rather than a larger working alphabet.

In the final comparison, we try to find the ideal values of the parameters b, b_f. Here we use the deterministic variant of 2LCRA and the architecture (Arch 3). The columns of the following tables correspond to different values of b and the rows correspond to values of b_f. Again, the values in the table record the number of found solutions in time t for the respective combination of b and b_f.

b_f \ b	1	5	10	20	30	50
1	251	389	345	232	179	117
5	382	425	347	237	181	114
10	434	427	357	242	179	122
20	496	444	348	225	173	118
30	509	408	340	235	166	118
50	513	397	325	229	178	119

Language L_1, $t = 10$ s

b_f \ b	1	5	10	20	30	50
1	7	43	33	55	59	57
5	13	47	49	52	55	33
10	20	52	40	33	37	22
20	31	37	30	33	28	27
30	30	43	31	35	21	16
50	35	19	10	24	14	11

Language L_3, $t = 50$ s

b_f \ b	1	5	10	20	30	50
1	1	11	26	29	36	36
5	1	20	37	35	33	30
10	7	34	24	28	25	25
20	21	29	21	32	26	14
30	30	24	19	14	17	23
50	38	25	11	13	10	11

Language L_5, $t = 100$ s

There does not appear to be a single best combination of these parameters. Indeed, the ideal setting might be different for each language. As a compromise, we take the values $b = 5$, $b_f = 10$.

In summary, we deem the following parameters to be the best choice for the learning algorithm:

- The deterministic variant of our model,
- the architecture (Arch 3) and
- the search parameters $b = 5$ and $b_f = 10$.

Note that we have arrived at this conclusion with the assumption that the parameters can be tweaked separately, which might not be the case. There might exist a better configuration that could not be discovered using our method.

6.2 Quality of the inferred automata

So far, we have only concerned ourselves with finding any solution, not inspecting its quality. Using the parameters established in the previous section, we now test the ability of our algorithm to produce robust, generalizing automata. Our testing was done using the following procedure.

For each language $L \in \{L_1, \ldots, L_6\}$ we generated a test data set $\langle S^+, S^- \rangle$ of 1000 positive and 1000 negative samples of pictures of varying sizes (k, l) for $k, l \in \{10, \ldots, 50\}$ (except for L_6 where the width is fixed) and then we performed 10 measurements for each $n \in \{10, 20, 50, 100\}$ consisting of

- generating a training set $\langle S^+_{\text{tr}}, S^-_{\text{tr}} \rangle$, $|S^+_{\text{tr}}| = |S^-_{\text{tr}}| = n$ with pictures of sizes (k, l) for $k, l \in \{5, \ldots, 10\}$,

- inferring an automaton $\mathcal{M}$ from $\langle S^+_{\mathrm{tr}}, S^-_{\mathrm{tr}} \rangle$ and
- measuring its performance on $\langle S^+, S^- \rangle$ as

$$x = \frac{|S^+ \cap L(\mathcal{M})| - |S^- \cap L(\mathcal{M})|}{1000} \quad .$$

The values of x range from -1 (indicating that $\mathcal{M}$ accepts the complement of L) to 1 (indicating that $L(\mathcal{M}) \approx L$).

From the ten measured values of x we computed their mean $\mathrm{E}(x)$ and standard deviation σ. The following table displays the results in the format $\mathrm{E}(x) \pm \sigma$.

L \ n	10	20	50	100
L_1	0.973 ± 0.033	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000
L_2	0.941 ± 0.090	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000
L_3	0.898 ± 0.094	0.911 ± 0.062	0.955 ± 0.037	0.953 ± 0.044
L_4	0.964 ± 0.032	0.999 ± 0.002	0.998 ± 0.005	1.000 ± 0.000
L_5	0.909 ± 0.076	0.968 ± 0.046	0.998 ± 0.003	1.000 ± 0.000
L_6	0.719 ± 0.273	0.804 ± 0.174	0.995 ± 0.015	0.998 ± 0.006

The results show that the inferred automata generally perform fairly well. By limiting the possible complexity of the automata we allow the learning algorithm to infer little else than the actual principle of the sampled languages.

In addition, we also include the following table of the average times of learning of the respective automata. These times correspond to a single-thread computation, so the expected times are m times lower when running m threads at once due to the aforementioned geometric distribution of the measurements. Specifically, on our testing machine with eight logical processors, the expected values are 8 times lower.

L \ n	10	20	50	100
L_1	0.045 s	0.075 s	0.193 s	0.391 s
L_2	0.037 s	0.066 s	0.183 s	0.496 s
L_3	0.628 s	2.285 s	5.126 s	10.764 s
L_4	12.706 s	34.546 s	78.140 s	214.852 s
L_5	3.189 s	15.507 s	71.781 s	45.225 s
L_6	13.252 s	49.714 s	65.549 s	206.650 s

When divided by the number of logical cores of a modern CPU, these times become quite bearable.

The results were achieved on a computer with *Intel Core i7 2600k* (4.25GHz) processor and 8GB RAM running *Microsoft Windows 8 x64* operating system.

6.3 Example outputs

The learning algorithm is designed to produce automata in a relatively comprehensible form of few rewriting rule patterns. The readability of the results is further augmented by the simplification procedure described in 5.2.4.

Below we show two examples of automata inferred by our algorithm using the deterministic model (M3) and the architecture (Arch 3) with $k = 2$. In both cases, the training set contained 50 positive and 50 negative examples.

The first example is an automaton inferred from examples of L_3. The automaton contains three rule patterns:

$$\begin{array}{ccc} & * & \\ * & \mathbf{a} & * \\ & * & \end{array} \rightarrow \begin{array}{ccc} & * & \\ * & \mathbf{a_1} & * \\ & * & \end{array}$$

$$\begin{array}{ccc} & \top, \mathbf{a_1} & \\ * & * & * \\ & \bot, \mathbf{a_1} & \end{array} \rightarrow \begin{array}{ccc} & \top, \mathbf{a_1} & \\ * & \mathbf{b_0} & * \\ & \bot, \mathbf{a_1} & \end{array}$$

$$\begin{array}{ccc} & * & \\ \vdash, \mathbf{a_1} & * & \dashv, \mathbf{a_1} \\ & * & \end{array} \rightarrow \begin{array}{ccc} & * & \\ \vdash, \mathbf{a_1} & \mathbf{b_1} & \dashv, \mathbf{a_1} \\ & * & \end{array}$$

The stars denote sets of all applicable symbols for the respective positions ($\{\mathbf{a}, \mathbf{b}\}$ for the central positions and $\mathcal{S} \cup \{\mathbf{a}, \mathbf{b}, \mathbf{a_1}, \mathbf{b_0}, \mathbf{b_1}\}$ elsewhere). The implicit weights of the symbols are 1 for the input symbols **a** and **b** and 0 for the auxiliary symbols $\mathbf{a_1}$, $\mathbf{b_0}$ and $\mathbf{b_1}$.

These patterns have a very clear interpretation — the second and the third pattern facilitate rewriting of horizontal and vertical lines, respectively. Note that the fourth pattern associated with the symbol $\mathbf{a_0}$ has been completely removed. This often happens during the simplification phase, when all symbols are removed from any set of admissible symbols in the pattern, rendering the pattern unusable.

The second example is an automaton learned from examples of L_4. The patterns are in the same format as above.

$$\begin{array}{ccc} & * & \\ \vdash, \mathbf{a_0} & \mathbf{a} & \mathbf{a}, \mathbf{b} \\ & * & \end{array} \rightarrow \begin{array}{ccc} & * & \\ \vdash, \mathbf{a_0} & \mathbf{a_0} & \mathbf{a}, \mathbf{b} \\ & * & \end{array}$$

$$\begin{array}{ccc} & * & \\ \mathbf{a_1}, \mathbf{b_1} & \mathbf{a} & * \\ & * & \end{array} \rightarrow \begin{array}{ccc} & * & \\ \mathbf{a_1}, \mathbf{b_1} & \mathbf{a_1} & * \\ & * & \end{array}$$

$$\begin{array}{ccc} & * & \\ \vdash, \mathbf{a_0} & \mathbf{b} & * \\ & * & \end{array} \rightarrow \begin{array}{ccc} & * & \\ \vdash, \mathbf{a_0} & \mathbf{b_1} & * \\ & * & \end{array}$$

Again, the patterns have an obvious interpretation. In each row of an input picture, the rules corresponding to the first pattern rewrite all **a**'s between the left border and the first **b** to $\mathbf{a_0}$. The third pattern's rules then rewrite this **b** to $\mathbf{b_1}$ and the second pattern's rules rewrite the following **a**'s to $\mathbf{a_1}$. If there is no **b** in the whole row, the automaton fails to rewrite the last **a**. Conversely, if there are two or more **b**'s in the same row, the second **b** is never rewritten and causes the computation to fail.

The shown examples are similar to the manually created automata for the respective languages, which makes them the best possible results. The imperfect average performance of the inferred automata measured in the previous section indicates that the outputs are not always so flawless, but they often are.

6.4 Inference of intersections

At the end of Chapter 5, we have outlined a technique for finding solutions in the form of intersections of automata. We tested this method on the language of permutations (described in 3.2), which can be defined as

$$L_{\mathrm{PERM}} = L_4 \cap L_4^{\mathrm{R}}.$$

As such, it can be learned as an intersection of languages of two automata with complexity limited by $k = 2$ (for architecture (Arch 3)).

With the same basic parameters as in the previous section (model (M3), architecture (Arch 3), $b = 5$, $b_f = 10$) we have performed ten measurements, where in each measurement we have generated a set of 50 positive samples of size $(5, 5)$ and 50 negative samples of sizes $(4, 5)$, $(5, 4)$ and $(5, 5)$, on which we have run the learning algorithm with the complexity parameter $k = 2$ and the maximum number of intersections set to $m = 2$. We measured the performance of the inferred automata (denoted by x) on a test set containing 1000 positive samples of size $(20, 20)$ and 1000 negative samples of sizes $(20, 19)$, $(19, 20)$ and $(20, 20)$, using the same formula for x as in the previous section.

From all ten measurements we have computed the average performance and the standard deviation, arriving at the values

$$\mathrm{E}(x) = 0.98, \sigma = 0.05.$$

The average time of finding one solution was 7.79 seconds. Unlike the averages in the previous section, this value is computed from the actual times needed to find a solution using eight threads running in parallel.

The language L_{PERM} can, in theory, be learned using a single automaton of the architecture (Arch 3) with the complexity parameter $k = 4$. In our attempt to do that, we have run the learning algorithm with similar training data as before in eight threads for one hour, in which a solution has not been found. The best result was an automaton rejecting 46 of 50 negative training samples, with test performance $x = 0.791$. Clearly, the concept of intersections significantly increases the ability of the algorithm to learn languages like L_{PERM}.

6.5 Limitations

The introduced algorithm, in its present form, is unable to learn some more complex picture languages in reasonable time. An example a of language that is difficult to learn is the language of forests described in 3.4.

The automaton accepting this language uses only four relatively simple rule patterns. However, the underlying concept of detecting a cycle in a graph is not so simple. The rewriting rules form a compact package that works well as a whole, and by changing or removing some rules, the rest ceases to function completely. It is therefore hard to learn this automaton using a local search algorithm that tries to find a solution through a sequence of minor improvements of one automaton.

This and other picture languages show us that there is certainly a room for improvement in our learning algorithm. They also suggest a possible direction for further research.

Conclusion

We had two main goals in this thesis. The first goal was to introduce a new model of a two-dimensional restarting automaton suitable for learning. We proposed a model called two-dimensional limited context restarting automaton, which is a simplified version of the restarting tiling automaton. Surprisingly, our model turned out to have the same power as the sgraffito automaton, which operates on a wholly different principle. In addition, we have defined the correctness preserving variant of our model which is equivalent to the deterministic sgraffito automaton. In contrast with the sgraffito automaton, our model allows for a comparatively easy and intuitive definition of many automata accepting commonly used picture languages. It also provides an alternative approach for the study of the shared class of accepted languages.

Because of the equivalence with sgraffito automata, the class of languages accepted by our model has rather well defined properties and comparisons with other classes of languages. An unsolved question is whether our model can be as powerful as restarting tiling automata using some limited scanning strategy. A second open question pertains the correctness preservation — it is yet to be seen whether it is indeed an undecidable problem to verify this property for an arbitrary automaton.

The second goal of this thesis was to present a learning algorithm for our model. As expected, the problem of learning of picture languages has proven itself to be a very challenging one. We have introduced a concise format for representing limited subclasses of our model and then we presented a modification of a local search algorithm suitable for learning thus represented automata. We have implemented and tested this algorithm and we have found that it is capable of learning some basic picture languages.

Our algorithm is not able to learn more complex picture languages in reasonable time, but, being the first of its kind, it can perhaps serve as a basis for future development in this unexplored field of study.

Overall, we can say that all of the goals outlined for this thesis were met in a satisfactory way.

Bibliography

[1] M. Anselmo, D. Giammarresi, M. Madonia. A computational model for tiling recognizable two-dimensional languages. *Theoretical Computer Science Vol. 410-37*, pp. 3520–3529, Elsevier, 2009.

[2] J. Barták. Recognition of picture languages. Master thesis, Faculty of Mathematics and Physics, Charles University, Prague, 2006.

[3] R. Barták. Constraint Programming: In Pursuit of the Holy Grail. *Proceedings of the Week of Doctoral Students (WDS99), Part IV*, pp. 555-564, MatFyzPress, 1999.

[4] S. Basovník. Learning Restricted Restarting Automata using Genetic Algorithm. Master thesis, Charles University, Faculty of Mathematics and Physics, 2010.

[5] M. Blum, C. Hewitt. Automata on a two-dimensional tape. *IEEE Symposium on Switching and Automata Theory*, pp. 155–160, 1967.

[6] A. Cherubini, M. Pradella. Picture Languages: from Wang tiles to 2D grammars. *Lecture Notes in Computer Science Vol. 5725*, pp 13-46, Springer, 2009.

[7] P. Černo. Grammatical inference of lambda-confluent context rewriting systems. *Vol. 294 of books@ocg.at*, pp. 85-100, Österreichische Computer Gesellschaft, 2013

[8] D. Giammarresi. Exploring Tiling Recognizable Picture Languages to Find Deterministic Subclasses. *International Journal of Foundations of Computer Science Vol. 22, No. 7*, pp. 1519-1532, 2011.

[9] D. Giammarresi, A. Restivo. Recognizable picture languages. *International Journal Pattern Recognition and Artificial Intelligence Vol. 6, No. 2 & 3*, pp. 241-256, 1992.

[10] C. de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010.

[11] K. Inoue, A. Nakamura. Some properties of two-dimensional on-line tessellation acceptors. *Information Sciences Vol. 13*, pp. 95–121, Elsevier, 1977.

[12] P. Jančar, F. Mráz, M. Plátek, J. Vogel. Restarting automata. *Lecture Notes in Computer Science Vol. 965*, pp 283-292, Springer, 1995.

[13] P. Jančar, F. Mráz, M. Plátek, J. Vogel. Different Types of Monotonicity for Restarting Automata. *Lecture Notes in Computer Science Vol. 1530*, pp 343-354, Springer, 1998.

[14] F. Mráz, F. Otto. Extended Two-Way Ordered Restarting Automata for Picture Languages. *Lecture Notes in Computer Science Vol. 8370*, pp 541-552, Springer, 2014.

[15] F. Mráz, F. Otto. Ordered Restarting Automata for Picture Languages. *Lecture Notes in Computer Science Vol. 8327*, pp 431-442, Springer, 2014.

[16] F. Mráz, F. Otto, M. Plátek. Hierarchical Relaxations of the Correctness Preserving Property for Restarting Automata. *Lecture Notes in Computer Science Vol. 4664*, pp 230-241, Springer, 2007.

[17] D. Průša, F. Mráz. New Models for Recognition of Picture Languages: Sgraffito and Restarting Tiling Automata. *Research Reports of CMP, Czech Technical University in Prague, No. 8*, 2012.

[18] D. Průša, F. Mráz. Two-Dimensional Sgraffito Automata. *Lecture Notes in Computer Science Vol. 7410*, pp 251-262, Springer, 2012

Appendix A

User Guide

In this chapter we provide a concise guide for using the attached program implementing the learning algorithm. The application provides an interface for managing picture sample sets, manual design of limited-context restarting automata, simulation of these automata over input pictures, procedural generation of sample sets of several picture languages and, last but not least, learning of automata from positive and negative samples through our proposed learning algorithm.

First we explain the installation of the program, then we describe the user interface.

A.1 Installation

The target platform of the program is *Microsoft Windows XP SP3* or higher. The application requires *Microsoft .NET Framework 4* to run. This framework can be installed using the download tool supplied on the attached CD in the `dotNET4` folder.

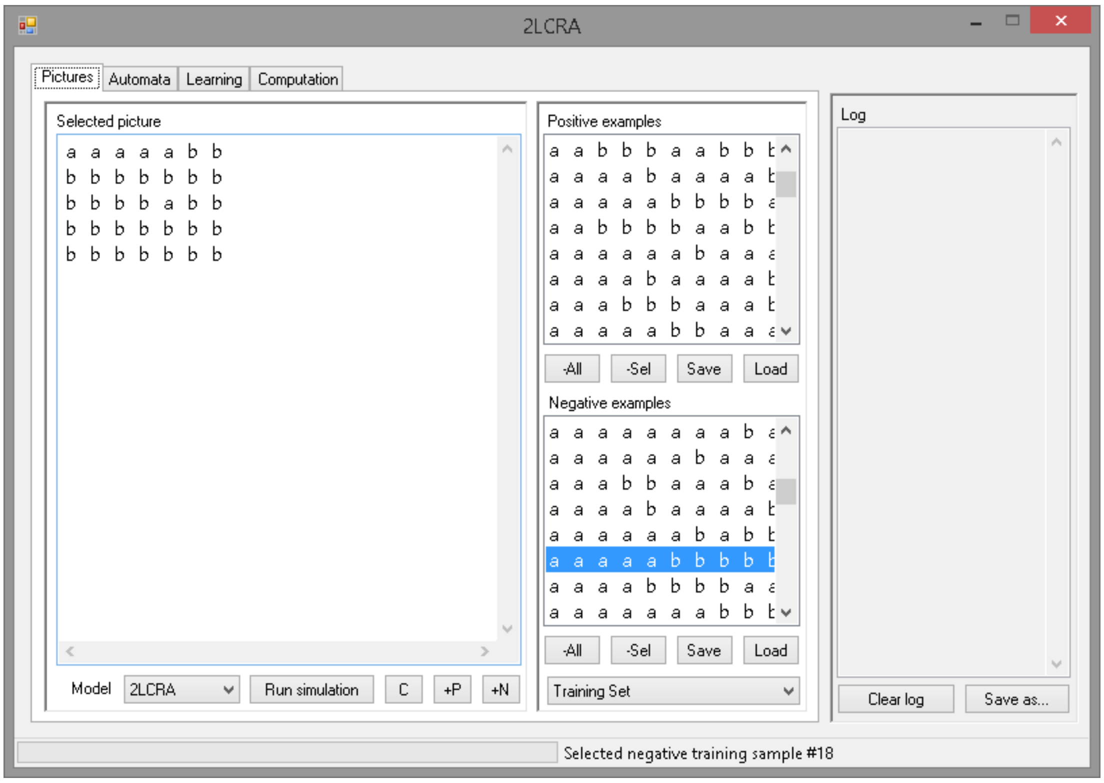
The main binary file of the application, `2LCRA.exe`, can be found on the attached CD in the folder `program\bin`. The program does not require any installation, it can be run either directly from the CD or from a local copy of the binary file.

The program's *C#* source files can be found in the `program\source` folder. The supplied project files are compatible with *Microsoft Visual Studio 2010* or newer. As the program is only a supplement of this thesis, we reduce the source code documentation to the form of commentaries within the code.

A.2 Using the application

In this section we explain how to use the application itself. The graphical user interface offers various controls, which are split into four tabs, and a log that displays results of the performed operations. We now describe each of the four tabs separately. At the very end we describe the format for storing the program's input data in files.

A.2.1 Pictures



Selected picture

```
a a a a a b b
b b b b b b b
b b b b a b b
b b b b b b b
b b b b b b b
```

Positive examples

```
a a b b b a a b b
a a a a b a a a a
a a a a a b b b b
a a b b b b a a b
a a a a a a b a a
a a a a b a a a a
a a a b b b a a a
a a a a a b b a a
```

Negative examples

```
a a a a a a a a b
a a a a a a b a a
a a a b b a a a b
a a a a b a a a a
a a a a a a b a b
a a a a a b b b b
a a a a b b b b a
a a a a a a a b b
```

Model: 2LCRA

Training Set

Selected negative training sample #18

This tab serves for managing training and test sets of picture examples for learning as well as for inputting a single picture for an automaton simulation.

Selected picture In the box on the left, you can edit a single picture. Each line in the box corresponds to one row in the picture, and alphanumeric strings on each line separated by white spaces correspond to single symbols. With the displayed picture you can, using the controls below, do one of the following:

Run simulation simulates the automaton defined in the **Automata** tab over the selected picture. An appropriate recognition algorithm is chosen according to the model selected in the left drop-down menu — the general 2LCRA recognition algorithm described in 5.1, the algorithm for correctness preserving automata (2.2), or the deterministic simulation (5.2.5). If the **Selected picture** box contains no input, the simulation is run on the current training and test data sets. The result of the simulation is shown in the log on the right.

C clears all symbols from the picture.

+P, +N adds the picture as a positive or as a negative sample to the current data set (training or test).

Examples The boxes on the right display the positive and negative samples of pictures that are contained in the data set selected below — either the training set used for inference of automata or the test set used for verifying their performance. Each box offers four operations:

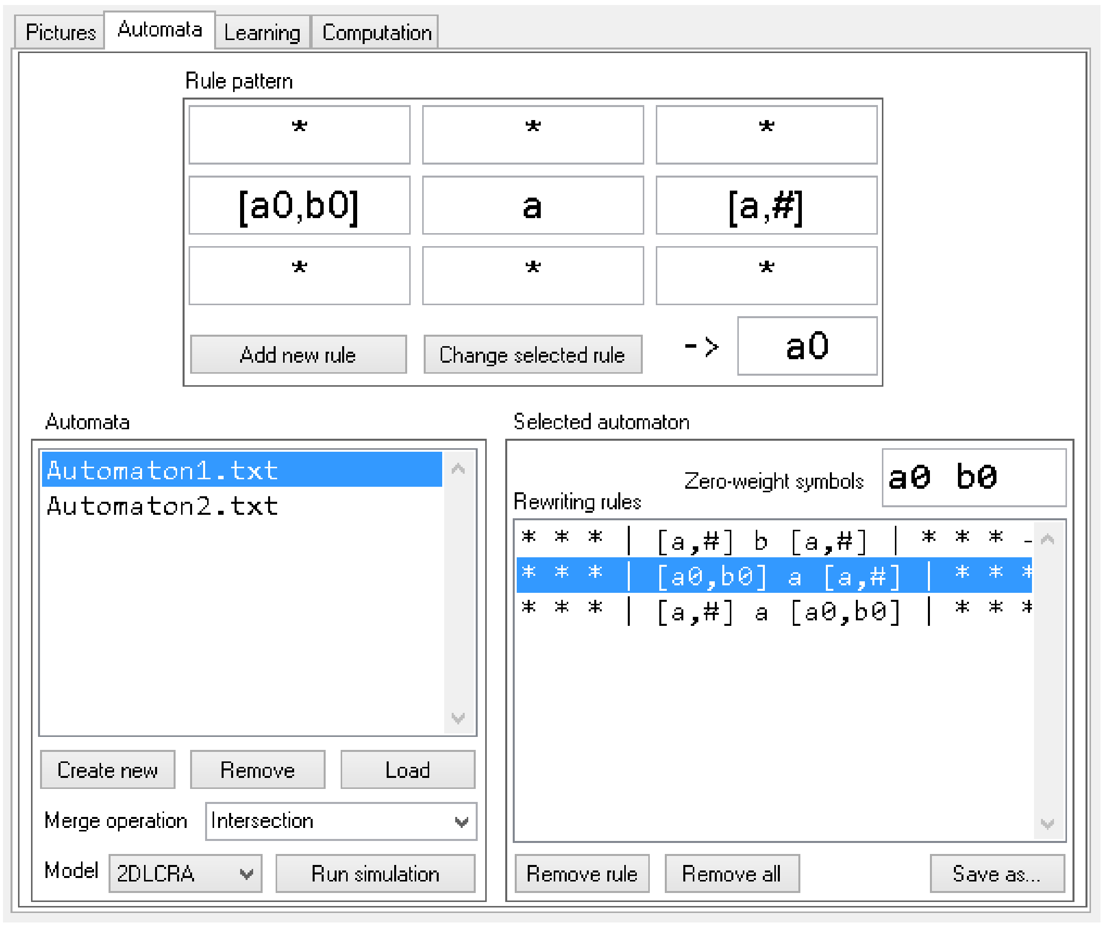
-All removes all pictures from the respective collection.

-Sel removes the selected picture from the list.

Save writes the picture set in a selected file, using a text format described later in A.2.5.

Load reads a set from a selected file using the same text format.

A.2.2 Automata



Rule pattern

*	*	*
[a0,b0]	a	[a,#]
*	*	*

-> a0

Automata

Automaton1.txt
Automaton2.txt

Merge operation: Intersection

Model: 2DLCRA

Selected automaton

Zero-weight symbols: a0 b0

Rewriting rules

```
* * * | [a,#] b [a,#] | * * * -
* * * | [a0,b0] a [a,#] | * * *
* * * | [a,#] a [a0,b0] | * * *
```

In this tab, you can manage automata and use them for simulation over pictures defined in the **Pictures** tab. The automata inferred by the learning algorithm are also displayed here.

Automata The lower left box contains a current list of automata. Using the buttons below, you can add a new empty automaton to the list, remove the selected automaton, or load an automaton from a selected file that contains a definition in a format described below in A.2.5. You can also run a simulation of the current automata over the selected picture in the **Pictures** tab, using any of the three recognition algorithms. If there are more automata on the list, the simulation results are merged using the selected merge operation.

Selected automaton The lower right box displays the definition of the selected automaton. An automaton is defined by a set of rewriting rules and a list of zero-weight symbols (The alphabet and its weights are generated implicitly from this definition). The rewriting rules are described by the displayed list of rule patterns. This list can be modified using the pattern editor above or by removing one or all patterns using the buttons below. The automaton can be saved to a file using the format outlined in A.2.5.

Rule pattern In the upper box you can edit a selected rule pattern. The pattern (in the sense explained in Chapter 3) consists of a 3-by-3 array of sets of accepted symbols for each position and a resulting symbol. One set of accepted symbols is defined using one of the following expressions.

`*` A star denotes any symbol of the working alphabet or a sentinel.

`!` An exclamation mark denotes an empty set (which renders the pattern unusable).

`S` An alphanumeric string, interpreted as a name of a symbol, denotes a set containing only this symbol. The hash sign (`#`) is a special name that denotes any sentinel from $\mathcal{S}$.

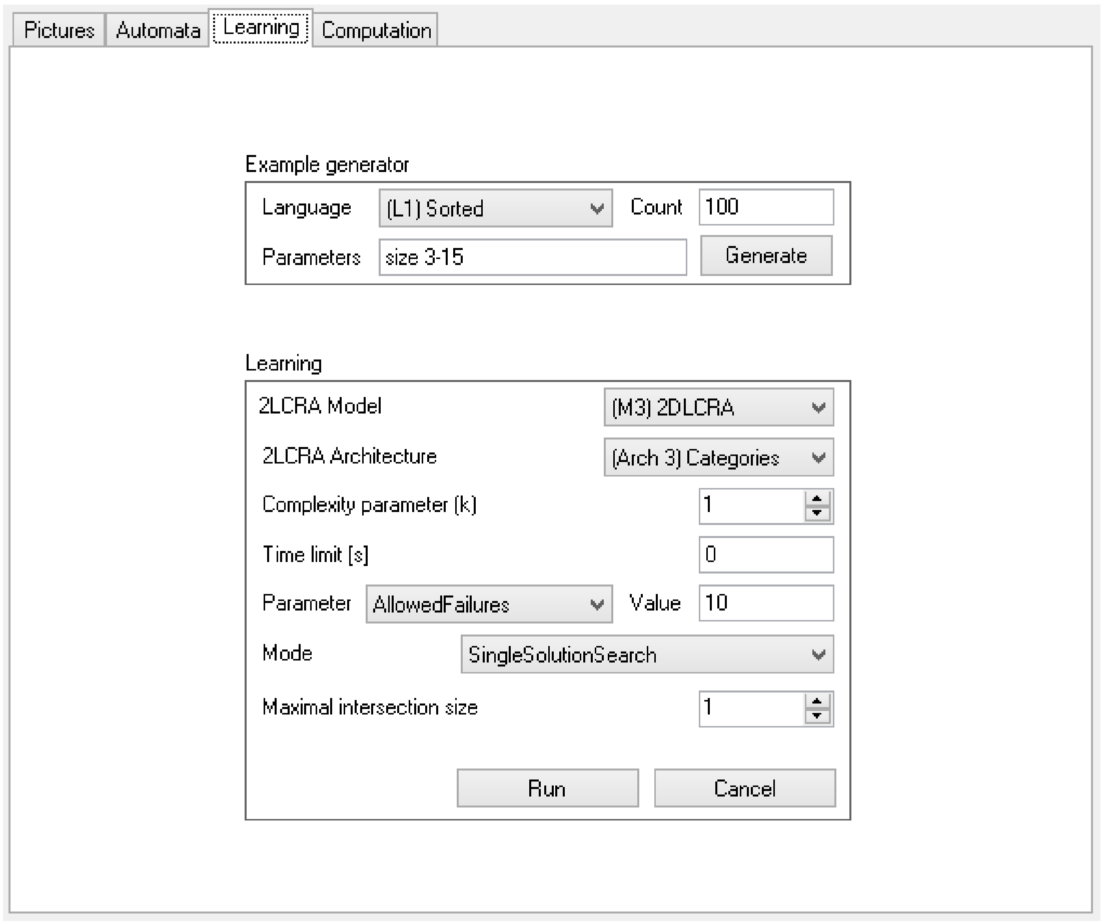
`[S1,S2,...,SN]` A list of symbol names in square brackets, separated by commas (without white spaces), denotes a set of the listed symbols.

`^expr` A caret denotes a negation of the following expression. This expression can be any of the above.

After defining a pattern, you can either add it as a new element of the list of patterns or use it to update the selected pattern.

Note that the deterministic 2LCRA model uses rules in the cross format (see 4.2.2), so the corner positions of the patterns are ignored during its simulation.

A.2.3 Learning



Example generator

Language: (L1) Sorted
Count: 100
Parameters: size 3-15
Generate

Learning

2LCRA Model: (M3) 2DLCRA
2LCRA Architecture: (Arch 3) Categories
Complexity parameter (k): 1
Time limit [s]: 0
Parameter: AllowedFailures
Value: 10
Mode: SingleSolutionSearch
Maximal intersection size: 1
Run
Cancel

In this tab you can generate examples of the predefined languages or launch the learning algorithm.

Example generator The upper panel serves for generating sample sets of languages. Upon clicking the **Generate** button, the data set selected in the **Pictures** tab (the training set or the test set) is populated by n positive and n negative examples of the selected language, where n is specified by the **Count** field. The **Parameters** field allows a text-based parametrization of the selected generator. The usual contents of this field is a string "`size X-Y`", where `X` is the minimal and `Y` is the maximal number of rows and columns of every generated picture.

The program source allows for relatively simple addition of new language generators. For reference, see the example class in the file

`ExampleGenerator.cs`.

Learning The lower box contains the controls of the learning algorithm. The basic parameters of the algorithm include a selection of the used model of automata (see 6.1), the architecture of automata (see 5.2.1) and the complexity parameter k (see 5.2.1). Additional parameters can be found in the drop-down menu:

AllowedFailures, StepWidth denote the parameters b_f, b of the search algorithm (see 5.2.3, 6.1),

StarWeight, ListWeight denote the parameters c_a, c_b of evaluation of automata (see 5.2.2).

A time limit can be imposed on every computation by setting the desired time, in seconds, in the corresponding field. After reaching the limit, the computation is stopped regardless of its result. A zero value means no time limit.

The algorithm can be run in the following three modes.

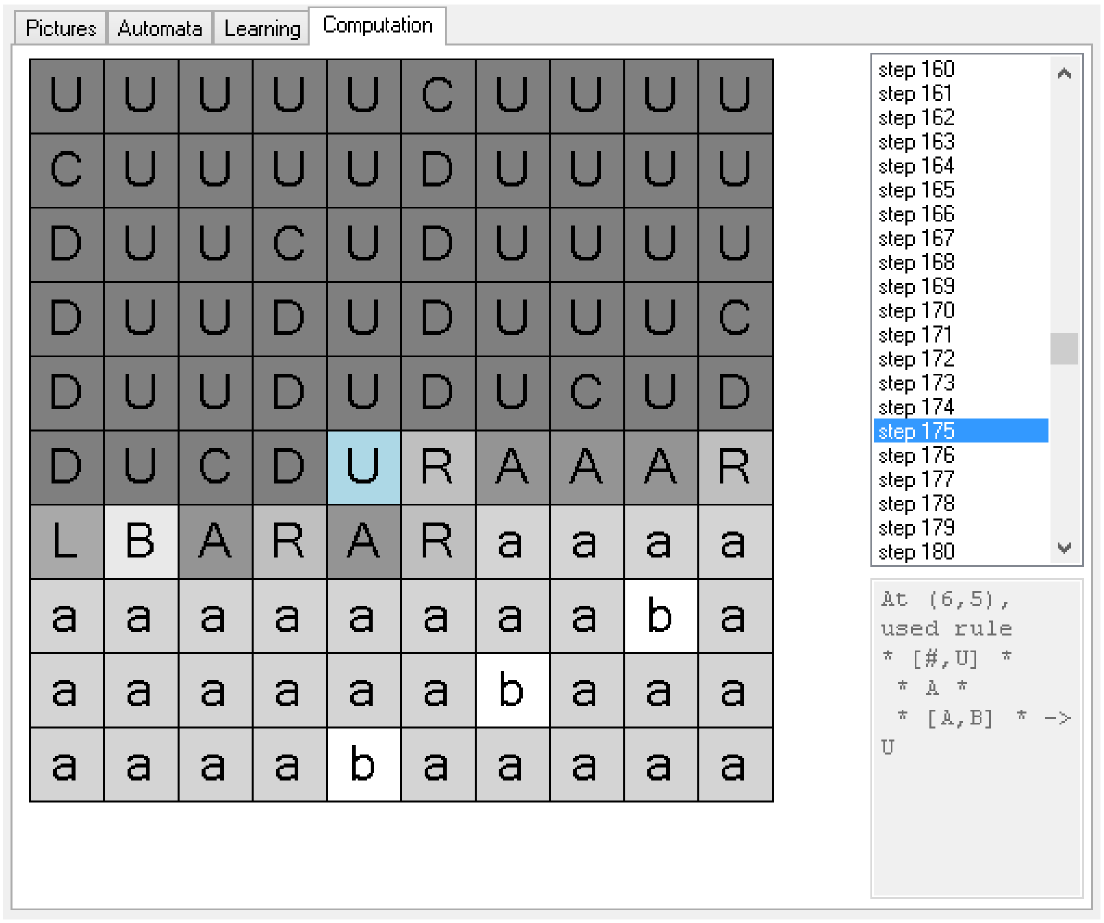
SingleSolutionSearch is the standard mode, in which all threads (whose number is determined automatically) try to learn an automaton from the training data set defined in the **Pictures** tab. When a solution is found, all threads are terminated and the inferred automaton is displayed in the **Automata** tab. Additionally, in this mode you can set a maximum intersection size parameter, which, if greater than one, causes the learning algorithm to look for a solution in the form of an intersection of automata (as described in 5.2.6).

SuccessCount is a mode where the computation does not stop upon finding a solution, but runs until it is explicitly canceled or the time limit is reached. The result of this mode is the count of the found solutions. This mode was used in Section 6.1.

MultipleDataTesting is a mode where the training data are split in n equally large parts (n is specified by the **Training data split** field) and the learning algorithm is run on all of them, one thread per part. The inferred automata are then tested on the whole test set defined in the **Pictures** tab. The result is an average performance of the automata on the test data. This mode was used in Section 6.2.

The results of the learning are displayed in the log on the right side of the program window.

A.2.4 Computation



U	U	U	U	U	C	U	U	U	U
C	U	U	U	U	D	U	U	U	U
D	U	U	C	U	D	U	U	U	U
D	U	U	D	U	D	U	U	U	C
D	U	U	D	U	D	U	C	U	D
D	U	C	D	U	R	A	A	A	R
L	B	A	R	A	R	a	a	a	a
a	a	a	a	a	a	a	a	b	a
a	a	a	a	a	a	b	a	a	a
a	a	a	a	b	a	a	a	a	a

step 160
step 161
step 162
step 163
step 164
step 165
step 166
step 167
step 168
step 169
step 170
step 171
step 172
step 173
step 174
step 175
step 176
step 177
step 178
step 179
step 180

```
At (6,5),
used rule
* [#,U] *
 * A *
 * [A,B] * ->
U
```

In this tab, you can explore the last simulation of an automaton performed over a single picture. This tab is activated automatically after running each simulation except for the case of rejecting a picture by a nondeterministic 2LCRA. By selecting a step of the computation in the upper right box you can visualize the state of the picture after performing all rewritings up to the selected step.

A.2.5 Input file format

Here we describe the used formats of files containing definitions of automata and sets of pictures. In both of these formats, a single symbol is represented by an alphanumeric string of arbitrary length.

Picture files

The picture files are used for storing sets of pictures defined in the **Pictures** tab. One file contains a list of pictures, separated by semicolons. Each picture's rows are defined by consecutive lines. Each line contains a list of symbols (one for each column) separated by spaces.

An example follows.

```
a b a b
a b a b
a a a a;

a b a
a a a;

a b
b a
a a
b a

a ab a a a a ba
a ab a a a ba a
a a ab a ba a a
a a ab a ba a a
a a a bab ab a a
a ba ba a a ab ab
ba a a a a a a;
```

Automata files

These files are used for storing definitions used in the **Automata** tab. One file contains a definition of a single automaton, given by a list of zero-weight symbols and a list of rewriting rule patterns.

The first line of the file contains a label "`zeros:`" followed by a space and a list of zero symbols separated by spaces, terminated with a semicolon.

Following that is a list of patterns separated by semicolons. Each pattern consists of eleven atoms separated by white spaces:

```
E11 E12 E13
E21 E22 E23
E31 E32 E33 -> R
```

`EIJ` are expressions defining the pattern, written in the format described in A.2.2. `R` is the resulting symbol of the pattern.

An example definition of an automaton accepting squares from $\{\mathbf{a}\}^{n,n}$ follows.

```
zeros: 1 d u;

# # #
# a *
# * * -> 1;

1 * *
* a *
* * * -> 1;

* * *
[u,1] a *
* [u,1] * -> u;

* [d,1] *
* a [d,1]
* * * -> d;
```